Karlsruhe Institute of Technology

Restartable Microkernels using Persistent RAM, Resilient Heaps, and Rust

Master's Thesis
submitted by

cand. inform. Philipp Oppermann

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Prof. Dr. Frank Bellosa

March 07, 2019 – September 06, 2019

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, September 6, 2019

Abstract

Non-volatile random access memory (NVRAM) is a promising memory technology that combines the performance and byte-addressability of DRAM with the non-volatility of hard disks and SSDs. With the increasing availability of NVRAM in both server and consumer systems, different approaches to utilize NVRAM in applications emerge.

In this work, we explore the use of NVRAM to make operating systems restartable after power outages and hardware failures. By keeping operating system state across reboots, the reliability of the system can be improved. For example, data loss can be prevented by continuing interrupted file operations after a power outage. The result of our work is a NVRAM framework that allows to safely add restartability to both the kernel and userspace programs.

Our framework uses a persistent heap to keep selected state across reboots. Since partial write operations to NVRAM might lead to soft errors, we make the heap resilient by using error correcting codes that ensure value consistency. To ensure a correct usage of our framework and rule out programmer errors at compile time, we utilize the type system of the Rust programming language. Notably, we ensure that restored values cannot contain pointers to non-restored values because they would become dangling after a restart.

Using our framework, we were able to make parts of the state of the *Redox* operating system restorable. To provide a foundation for restartable file systems and I/O staging, we implemented support for persistent file descriptors. We were also able to use the error correcting codes of our framework to considerably improve the bit-flip resilience of the kernel-internal list of processes and threads.

Contents

Chapter 1

Introduction

Non-volatile random-access memory (NVRAM) has become more available in recent years. Through technologies like Intel Optane [1] and Viking NVDIMM [2], NVRAM is now available to enterprise and even consumer computers. It is mainly used for improving I/O performance, for example through I/O staging [3] in high performance or cloud computing.

Another use case of NVRAM is the optimization of in-memory databases [4]. By keeping the database state in non volatile RAM, downtimes after power outages can be significantly reduced. Compared to DRAM, the higher supported capacity and the lower cost of NVRAM also enable larger in-memory database sizes and thus improve performance. To efficiently utilize NVRAM for database systems, Lersch et al. [5] integrate it into the transactional storage manager of the database, treating it as main memory and storage at the same time.

In this work, we explore how NVRAM can be used for improving the robustness of microkernel operating systems against random reboots. The goal of such restartable systems is to reasonably continue execution after a power outage or a hardware failure. For example, the system could finish important I/O operations to prevent loss of data. The result of our work is a NVRAM framework that can be used from both the kernel and userspace programs.

The fundamental idea of our approach is to keep important operating system state in NVRAM. This way, the state is not lost when the system is suddenly stopped and can be restored when the system is started again. The difficulty is that various consistency problems can occur in this process. First, non-atomic writes to the state can be interrupted in between, so that the state is inconsistent when the system powers down. Second, bit-flips can occur in the stored data, also leading to inconsistent state in the restore process. The probability and characteristics of such bit-flips depend on the used NVRAM technology, but no technology can completely rule them out. Third, the restored state might contain pointers to non-restored state, which leads to dangling pointers.

To prevent the mentioned consistency problems, our solution utilizes resilient data structures and the type system of the Rust programming language. Resilient data structures employ *error correcting codes* to prevent consistency errors caused by bit-flips. They can also provide a limited form of protection against partial writes, depending on the used error correcting code. For full protection against partial writes, software transactional memory can be used (left for future work). To prevent dangling pointers because of partially restored state, our solution employs the type system of the Rust programming language to guarantee that restored values cannot contain pointers to non-restored values.

In order to improve usability, our framework provides a `#[restorable]` attribute that makes a static variable restorable with minimal boilerplate. The attribute is implemented using the *procedural macro* feature of Rust and requires no compiler modifications. Further, we provide types that combine existing wrapper types with an error correcting code. By providing the exact same interface as the original wrapper type, resilience can be added to wrapped types with a minimal number of code changes.

Using our framework, we were able to make parts of the state of the *Redox* operating system restorable after reboots. Namely, we implemented support for persistent file descriptors that stay valid across reboots. Persistent file descriptors make it possible to use the NVRAM for implementing resilient staging I/O. For example, write requests could be staged to NVRAM until they are written to disk so that they could be restarted after a sudden power outage.

We were also able to use the error correcting codes (ECCs) of our framework to make the kernel-internal list of processes and threads of Redox resilient against bit-flips. By combining existing wrapper types with error correcting codes without changing the public interface, only a minimal amount of code required modifications for implementing the ECC-protection. Through fault injection, we show that the error correcting codes considerably improve the resilience against bit-flips.

The rest of this chapter gives an introduction to non-volatile memory, restartable kernels, memory consistency, kernel resilience, and NVRAM frameworks. The following chapters are structured as follows. Chapter 2 provides background information about the Rust programming language and the Redox operating system. Chapter 3 describes the high-level design of our framework and explains the provided components and their interface. Chapter 4 describes selected implementation details of the fundamental components and shows how our framework can provide its safety guarantees. In Chapter 5, we evaluate our framework by using it on parts of the Redox operating system. Finally, we conclude our work in Chapter 6.

1.1 Non-Volatile Random Access Memory

Traditional types of random access memory (RAM) are volatile, which means that the stored data is lost when the memory is powered off. Dynamic RAM (DRAM), which is commonly used as main memory, stores data in capacitors that need to be periodically refreshed to hold the data. In contrast, static RAM (SRAM) stores each data bit in a flip-flop that has two stable states and needs no periodic refreshing, but still requires the supply voltage to keep this state. SRAM is faster than DRAM, but it is more expensive, has a lower cell density, and a higher operational power consumption. Therefore most modern computers use SRAM only for CPU caches, where a high performance is most important.

To keep data while the system is powered off, it must be written so some kind of non-volatile memory that is able to keep the state without power supply. Common types of non-volatile memory include hard disk drives, solid-state drives, and CDs/DVDs. Traditional hard disks use rotating, magnetic disks and a moving actuator to retrieve and change the magnetization of disk areas. Solid state drives instead use flash memory for storing the information, which is faster and requires no moving parts. CDs and DVDs use a laser diode to read (and write) microscopic structures to a plastic disk coated with a thin layer of aluminum.

The disadvantage of all mentioned non-volatile memory types is that they only support block-wise addressing of memory, while RAM allows to read and write individual bytes. They are also much slower than DRAM and SRAM and do not support uniform random access. For example, a sequential read on a hard disk is considerably faster than reading random data blocks. These restrictions make the mentioned non-volatile memory types unsuitable as main memory, so that they are commonly only used as secondary memory.

To combine the advantages of fast, byte-addressable RAM and non-volatile secondary memory, different kinds of *non-volatile RAM* are in development [6]:

- **Phase Change RAM (PRAM):** Phase change RAM [7] changes the resistance of chalcogenide glass, which is a material that is also used for writable CDs and DVDs. Because of long quench times, PRAM requires write times of up to 300ns and has a large power consumption on writes [8]. PRAM storage is already available on the market, for example as Intel's *3D XPoint* [9, 10]. Other companies that use the technology include Samsung, IBM, and Western Digital.

- **Resistive RAM (ReRAM):** Resistive RAM [11] works by changing the resistance of a dielectric material. It is still in early development, but shows promising results regarding to write performance and power consumption [8]. Companies researching ReRAM include Viking [12], Panasonic [13] and HP [14, 15],

- **Ferroelectric RAM (FeRAM):** Ferroelectric RAM [16] has a similar construction as DRAM. The difference is that instead of normal capacitors, capacitors with a ferroelectric layer are used, which can hold an electric polarization without power supply. Compared to flash memory, FeRAM uses less power and has a higher write performance. However, is has lower storage densities and higher cost. FeRAM was developed by Ramtron International and licensed by Texas Instruments and Fujitsu.

- **Magnetoresistive RAM (MRAM):** Magnetoresistive RAM [17] uses magnetic domains for storing data. There are two newer variants of MRAM, *spin-transfer torque* (STT-MRAM) [18] and *thermal-assisted switching* (TAS-MRAM) [19]. STT-MRAM uses spin-aligned electrons to torque the magnetic domains. TAS-MRAM uses quick heat-ups of the magnetic tunnel junctions. MRAM is used by Samsung, GlobalFoundries, and Everspin Technologies (already in production).

- **Millipede Memory:** Millipede Memory [20] works by burning nanoscopic pits into the surface of a thin polymer layer. Millipede Memory is developed by IBM, but no commercial products exist yet.

Several of the mentioned technologies are already available on the market and used in production, for example Intel Optane [1] and Viking NVDIMM [2]. The unique properties of non-volatile RAM make it a promising technology for a wide range of applications.

1.2 Restartable Kernels

System crashes and power outages are a serious problem for high-availability systems since they often result in data loss and long reboot times. For this reason, ongoing research tries to make systems restartable, either in part or fully. The goal is to keep critical data across reboots to reduce the impact of a system crash.

With the goal of improving file system performance, Chen et al. [21] propose caching file modifications in (volatile) RAM across hot reboots instead of using write-through caching. By adding protection to the data in memory, expensive periodic write-backs to disk are no longer needed to achieve reliability, which results in improved performance.

Sundararaman et al. [22] propose an approach to make the file system restartable after runtime CPU exceptions in order to avoid inconsistent state and data loss. The idea of the approach is to create regular checkpoints of the file system and keeping logs of all file system operations, heap allocations, and acquired locks.

Depoutovitch et al. [23] propose introducing an additional *crash kernel* that allows applications to survive kernel crashes. The idea is that the crash kernel takes over when the main kernel panics. It reads the state from the main kernel and launches crash routines that restore a working environment for applications.

Recursive restartability [24] describes the property that a system consists of individual restartable components. Together these components form a restart tree that makes restarts of the complete system possible. A system that fulfills the recursive restartability property has the advantage that it can often recover from so-called *Heisenbugs* such as race conditions or heap corruptions, which are difficult to debug and reproduce. Recursive restartability can also considerably improve the recovery time after crashes [25].

Narayanan et al. [26] propose to make the whole system persistent by using only non-volatile memory instead of DRAM. Transient state in processor registers and caches is flushed to NVRAM on a system failure using the residual energy from the system power supply. While this approach minimizes required code modifications in applications, it requires adjusting all device drivers to reinitialize the corresponding devices to their previous state after a reboot.

1.3 Memory Consistency

One of the challenges of creating a restartable kernel is to keep it in a consistent state across reboots. This section gives an overview of the different types of memory consistency. It then explains common causes for consistency violations and discusses additional consistency challenges that are unique to restartable kernels.

1.3.1 Types of Consistency

Consistency here means the *integrity* of a value. There are three different levels of consistency that build onto each other. The fundamental consistency class is *bit-wise consistency*, which ensures that a stored value does not change randomly. *Pointer consistency* additionally guarantees that pointers to other memory locations stay valid. Finally, *semantic consistency* ensures that a value stays compatible with other values without violating application specific invariants.

Bit-wise Consistency

Bit-wise consistency ensures the fundamental invariant that a written value retains its bit representation until it is overwritten with a different value. This property is essential for a computer system because programs need to trust their own data. For example, a single Boolean variable that randomly changes its value could result

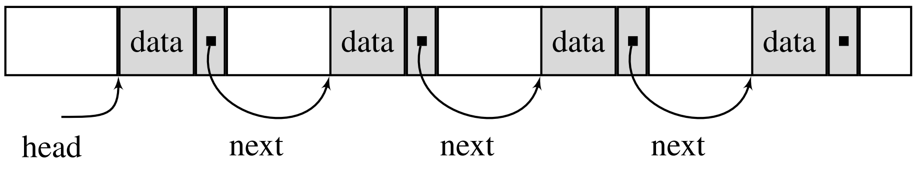

Figure 1.1: A Linked List Data Structure using Pointers

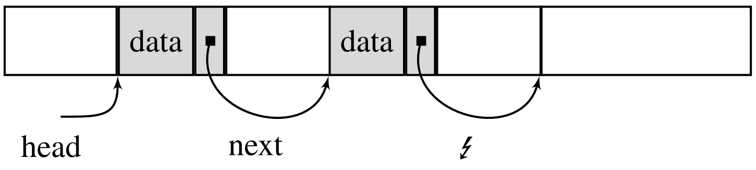

Figure 1.2: A Linked List Data Structure with a Dangling Pointer

in vulnerabilities (e.g. a flipped `is_admin` flag), incorrect behavior (e.g. endless loops), or complete system crashes (e.g. by misconfiguring a hardware device). Since the program code itself is stored in memory too, it could also change and lead to the arbitrary behavior of the program.

Pointer Consistency

Pointers are a common data type that refer to other values by storing their address. As an example, Figure 1.1 shows how pointers can be used to construct a linked list data structure. Each node stores a data item and a pointer that points to the memory location of the next node.

Pointer consistency ensures that all pointers point to valid values at all times. This is an important property because an invalid pointer can lead to undefined behavior of the program. For examples, the linked list structure in Figure 1.2 contains a pointer to an unused part of the memory. The content of this memory location is undefined, so traversing the list can lead to arbitrary results depending on the memory content.

A bit-wise consistent value can still violate pointer consistency, for example when the pointed value no longer exists. However, without bit-wise consistency, pointer consistency is violated too since pointers can change arbitrarily and point to undefined memory addresses. For this reason, pointer consistency is a subset of bit-wise consistency.

Pointer consistency is often mentioned as a part of *memory safety*, which additionally rules out other classes of vulnerabilities such as buffer overflows or reading uninitialized data. While most garbage collected languages such as Java or Python are memory safe by default, system programming languages such as C or C++ are often not. This leads to a large number of vulnerabilities for systems software in practice. For example, about 70% of the vulnerabilities that Microsoft assigns a CVE number are memory safety issues each year [27].

Semantic Consistency

Even when a value is both bit-wise and pointer consistent, it can still violate invariants of the program. An example is a linked list node that points to itself and thereby leads to a loop when traversing the list. We call the property that a value fulfills all program specific, higher level invariants of the program *semantic consistency*.

While a violation of bit-wise or pointer consistency can lead to undefined behavior, the consequences of a semantic consistency violation are more delimitable. They can still cause invalid program behavior, but they do not violate the integrity of the program itself.

1.3.2 General Consistency Violations

This section describes common types of consistency violations that are not specific to restartable kernels. At the hardware level, *unreliable memory* can lead to flipped bits and thus violations of bit-wise consistency. In software, errors in manual memory management can lead to *dangling pointers* that violate pointer consistency. Concurrency errors in multithreaded programs can cause *data races* that lead to consistency violations and undefined behavior.

Unreliable Memory

Even though modern hardware is very reliable, both soft and hard errors can still happen. Examples for hard errors are the failure of a hard disk or a broken CPU fan that causes a system shutdown. Soft errors are for example bad RAM blocks that are no longer able to store data. In contrast to hard errors, soft errors are much more dangerous because they often remain unnoticed. They can silently corrupt stored data and violate bit-wise consistency, which in turn can lead to undefined behavior.

Unfortunately, soft errors cannot be fully prevented in main memory. Even with fully intact memory hardware, environmental radiation can lead to soft errors in form of flipped bits [28]. While such errors occur infrequently enough to be ignored by normal applications, they are relevant for large scale server applications [29]. In contrast to secondary storage, standard RAM typically uses no checksums to guarantee the consistency of the stored data. While there exists special ECC-RAM that includes such a checksum at hardware level, it is not always used in server systems due to its high cost and the rareness of soft errors. Even with ECC-RAM, only a limited number of bit-flips can be detected and/or repaired, depending on the used checksum.

For non-volatile memory (described in Section 1.1), soft errors become more relevant. The reason is that many non-volatile memory techniques are susceptible to bit-flips. For example, Phase Change RAM suffers from resistance drifts, write noise, and spontaneous crystallization [30, 31]. With resistive RAM, sneak currents [32] and so-called *LRS* and *HRS* retention failures can occur [33–35]. For this reason, systems that use NVRAM need to protect the stored data from soft errors.

To harden critical applications and data against unreliable memory, several approaches exist to make them *resilient* against flipped bits. A common idea is to add redundancy to important data at the software level, for example in form of a checksum. This redundancy makes it possible to detect and recover from a limited number of bit-flips. While the protection of userspace applications is a well researched problem, ongoing work explores techniques to also harden the operating system kernel. We give an overview of this work in Section 1.4.

Dangling Pointers

Dangling pointers are pointers that no longer point to valid memory locations. Apart from unreliable memory, dangling pointers mostly occur because of *use-after-free* vulnerabilities. Such a vulnerability occurs when a heap-allocated value is freed, but there still exists a pointer to it. On subsequent uses of that pointer, undefined behavior occurs because pointer consistency is violated.

An attacker can often exploit dangling pointers to cause denial of service, privilege escalation, or even execute arbitrary code. For example, a recently found use-after-free vulnerability of the networking subsystem of the Linux kernel allowed a local attacker to take full control of the system [36].

To prevent such vulnerabilities, several approaches exist. For example, *Undangle* [37] is a runtime system to detect dangling pointers before they are used. *GUEB* [38] performs a static analysis of disassembled binary code to detect use-after-free patterns. *Fuzzing* [39] is a testing technique that provides randomized input data to a program and monitors it for crashes. Tools like *syzkaller* [40] apply fuzzing to find issues in the system call interface of operating system kernels.

While these approaches allow the detection of vulnerabilities in some cases, only memory safe programming languages can guarantee the complete absence of dangling pointers. For this reason, memory safe *system* programming languages such as Rust see a growing interest [27, 41, 42]. For example, the Linux kernel plans to add an optional, experimental framework for writing drivers in Rust [43].

Data Races

Multithreading allows programs to run multiple threads of execution concurrently. Depending on the number of CPU cores, the used operating system policies, and the number of active threads in the system, the threads can be executed on different cores in parallel or on the same core using time-slicing and preemption.

Since threads share the same address space, they can access the same variables concurrently. This can lead to consistency problems if the accesses are not synchronized properly, which is called a *data race*. For example, a data race can occur when two threads concurrently try to write to a large global variable. Each write is split into multiple CPU instructions, which can interleave with the writes from the other thread. The result is that parts of the written value are from one thread and parts from the other, which results in an inconsistent value. This violates bit-wise consistency, since the stored value is different from the values that were written.

Many compiled languages such as C++ treat data races as undefined behavior so that the effects are not limited to a single inconsistent value. Instead, the program can crash, loop endlessly, or exhibit any other behavior that was beneficial to the compiler when optimizing the program [44]. For this reason, data races are often close to impossible to debug and regularly lead to security vulnerabilities in practice [45].

To ensure consistency in multithreaded programs, all conflicting operations must be properly synchronized, for example by using locks for mutual exclusion. Alternatively, hardware support for atomic operations can be used together with an appropriate *memory ordering* [46] that specifies how the compiler and the CPU are allowed to reorder operations. For example, an atomic memory load operation in C++ with the *acquire* ordering forbids reordering any load or store operations before this atomic operation.

For formally modeling consistency in multithreaded programs, several models exist. The two most relevant models are the *happens-before* relation and the *communication sequential processes* model:

- The **happens-before relation** [47] is a partial ordering of the system events. A happens-before relation exist between two operations `A` and `B` if either `A` is sequentially executed before `B` in the same thread or `A` is send operation of a message and `B` is the corresponding receive operation for the same message. Further, the relation is reflexive and transitive. Operations with no happens-before relation are called *concurrent*.

  With this model, a data race occurs when two *concurrent* operations access the same memory location and one of them is a write operation. The happens-before relation is commonly used for data race detectors such as Helgrind [48] and ThreadSanitizer [49].

- The **communicating sequential processes (CSP)** model [50] describes parallel programs as processes that exchange messages through channels. The model influenced the design of programming languages such as occam [51], Go [52, 53] or Crystal [54, 55].

  By modeling programs in CSP (e.g. from a program trace) and generating all possible interleavings, it is possible to detect data races by searching for certain patterns in the CSP process [56]. This makes it possible to detect thread interleavings that lead to data races.

While there exist a large number of data race detectors such as ThreadSanitizer [49] or Go's data race detector [57], only a few solutions that are able to fully prevent data races exist. Boyapati et al. [58] present a static type system that prevents data races through an ownership system that requires to specify the required synchronization for each object. Using this type system, they present a race-protected Java-variant. Matsakis et al. [59] introduce a type system based on intervals, which are objects that represent the time in which a block of code will execute. Each object is assigned a guard object, which can either be an interval or a lock. By verifying the happens-before relation between intervals, data races can be prevented. The Rust programming language utilizes similar ideas in its borrow checker and `Send`/`Sync` traits and thus shows that data race prevention at compile time is feasible in practice.

1.3.3 Consistency Violations in Restartable Kernels

In addition to the general causes for consistency violations described in the previous section, there are new sources for consistency violations in restartable kernels. The reason is that the system can be stopped at an arbitrary time, which can lead to partial writes. Further, if only parts of the kernel state are restored after a reboot, violations of pointer and semantic consistency can occur even with memory safe programming languages.

Partial Writes

Most source code instructions are not *atomic*, which means that they are translated into multiple smaller operations. For example, Listing 1.1 shows a `test` function, which writes to a given array, and the corresponding assembly code generated by the compiler. The compiler translates the single write operation at the source code level into multiple assembly instructions, one for each array element.

```
pub fn test(array: &mut [u64; 3]) {
    *array = [1,2,3];
}
```

```
test:
mov qword ptr [rdi], 1
mov qword ptr [rdi + 8], 2
mov qword ptr [rdi + 16], 3
ret
```

Listing 1.1: Rust Source Code and Generated `x86_64` Assembly Code for an Example Function that Writes to an Array

Even small write operations can be split into multiple assembly instructions by the compiler, for example when the value is not properly aligned. As an example, Listing 1.2 shows the source code and corresponding ARM assembly code for a write to a non-aligned `number` field of type `u32`. The non-aligned field is forced by annotating a `Test` struct with the `#[repr(packed)]` attribute, which prevents any padding between the struct fields. This way, the `number` field only has an alignment of `1` instead of `4`, which is the required alignment for `u32` types on the ARMv6 architecture. As a result, the compiler needs to translate the single write operation at the source code level into four `strb` instructions that all write a single byte of the `number` field.

Depending on the CPU architecture, even single assembly operations are often not atomic. For example, the `pusha` instruction of the `x86` architecture, which pushes all general purpose registers to the stack, is split into multiple memory operations at the CPU level. Even the simple `inc` instruction of the `x86` architecture, which increments a single memory operand, is non-atomic [60].

Non-atomic operations can be interrupted in between, for example by a CPU interrupt or a system crash. Such an interruption can lead to a *partial write*, which means that the target value is (temporarily) left in an invalid state. Accessing the invalid value can lead to undefined behavior and cause system crashes.

Partial writes alone are not a problem. In preemptive multi-threading systems, partial writes happen regularly as each thread can be interrupted at an arbitrary time by timer interrupts and the scheduler. These partial writes do not lead to undefined behavior since the write operation is finished when the execution of the thread is resumed. Namely, no read operation of the value occurs before it is fully written (assuming no data races exist in the program). For the same reason, partial writes caused by power outages are not a problem in normal systems either. Even though the partially written value is not bit-wise consistent, it is never read again since the whole memory content is reinitialized when the system restarts.

```
#[repr(packed)]
pub struct Test {
    _byte: u8,
    number: u32,
}

pub fn test(test: &mut Test) {
    test.number = 42;
}
```

```
test:
mov r1, #42
strb r1, [r0, #1]!
mov r1, #0
strb r1, [r0, #3]
strb r1, [r0, #2]
strb r1, [r0, #1]
bx lr
```

Listing 1.2: Rust Source Code and Generated `ARMv6` Assembly Code for an Example Function that Performs a Non-aligned Write

The situation is different for restartable kernels because they try to restore values from NVRAM after a reboot. Since the CPU state and the stack content is lost on a power outage, the system cannot complete potential partial writes. To avoid undefined behavior, the system needs to either prevent partial writes or ensure that no partially written values are restored.

A common solution to this problem is the use of *software transactional memory* (STM) [61]. Similar to database transactions, STM combines multiple read and write operations into a single atomic transaction. Transactions are always fully applied so that no intermediate states become visible. In case of conflicts, for example when two threads try to perform a transaction at the same time, a transaction can be aborted and retried. Conceptually, STM is similar to atomic operations provided by the hardware, but it works for an arbitrary number of read and write operations and for arbitrary data sizes. The disadvantage of STM is a potentially high performance overhead.

Partial Restores

Restartable systems often only restore a selected subset of their state on a reboot. There are multiple reasons for this. First, the content of the CPU registers is lost on a power outage, which includes the current instruction and stack pointers, so that a restoration of the full state is not possible. While techniques such as early power outage detection might allow saving the whole system state, they are limited by hardware support and do not work for all types of crashes. Second, there are certain operations that need to be repeated after a reboot even if previous state is restored, such as reinitializing hardware devices. Third, the amount of NVRAM is often limited and its usage incurs performance overheads. For these reasons, restartable systems typically only perform a *partial restore*, which means that they restore a selected subset of the state and reinitialize the rest of the state as in a normal boot.

Since each restored value is restored in its entirety, no violation of bit-wise consistency occurs through the restore process. However, pointers contained in the restored value can become invalid if they point to non-restored values and thus violate pointer consistency. We solve this problem in our framework by permitting only values without potentially unsafe pointers in NVRAM.

In addition to pointer consistency violations, partially restored state can also lead to violations of semantic consistency. For example, a value might contain a process ID that is no longer valid or was assigned to a different process after reboot. Since semantic consistency is defined by the invariants of the applications itself, no general solutions to this problem are possible. Instead, the application itself must check the restored data for violated invariants. To support this, our framework allows applications to define their own consistency checks through custom wrapper types.

1.4 Kernel Resilience

As mentioned in Section 1.3.2, unreliable DRAM memory can lead to random bit-flips due to environmental radiation. Many upcoming NVRAM technologies also have reliability issues that can lead to flipped bits.

To increase the robustness of applications in the light of unreliable memory, techniques like checkpoint/restart [62] and replication [63] exist. Checkpoint/restart works by regularly creating checkpoints of a running program and resuming execution from a previous checkpoint when the program crashes (or becomes inconsistent). The idea of replication is to concurrently run multiple replicas of the program in separate address spaces and regularly compare their behavior, for example on each system call and CPU exception. This makes it possible to detect corrupted program instances using majority voting.

Approaches like checkpoint/restart and replication work well for protecting userspace programs, but need support from the operating system. For example, checkpoint/restart requires a disk or network driver to store checkpoints and replication requires the ability to run multiple processes and communicate with them. This makes the techniques unsuited for protecting the operating system kernel itself.

Since the kernel is much smaller than the application programs, it is less likely affected by soft errors. However, a soft error within the kernel can have more severe consequences because it can affect the whole system instead of only a single application. For this reason, there is ongoing research to make operating systems (and embedded programs) resilient too. Section 1.4.1 gives an overview of this research.

The common building block for achieving kernel resilience are *error correcting codes* that add redundancy to stored values. Section 1.4.2 introduces error correcting codes and presents common variants.

1.4.1 Related Work

With the goal of making operating system data structures resilient, Borchert et al. [64] present a software-based memory error protection approach that uses aspect-oriented programming to add error correcting codes to objects. In a follow-up study [65], they combine this approach with static program analysis to harden the L4/Fiasco.OC Microkernel with minimal runtime overhead.

Apart from resilient data structures, Stumpf [66] discusses additional methods to harden the kernel. Through asynchronous checks that verify both the checksums and semantic properties, memory corruption can be recognized early in order to minimize error-propagation. Other presented approaches include checkpoint/restart support for operating system services, replicated device drivers, and checksum-protected IPC messages.

Shafique et al. [67] present a multi-layer software reliability approach that combines aspect oriented programming with reliability-driven compilation and reliable operating systems. Reliability-driven compilation optimizes the generated code for reliability, for example by reducing the number of critical instructions such as load, store, or branching instructions or by adding redundancy to the instructions themselves. The reliable operating system provides a runtime reliability management mechanism that dynamically activates a redundant multithreading system for application tasks depending on their importance and the observed soft-error rates. For hardening the operating system kernel, the authors propose using pointer-less control flow structures, fine-grained assertions, and AN-encoding [68].

In more recent work, Santini et al. [69] evaluate the radiation reliability of the two dependability-oriented real-time operating systems *eCos* and *dOSEK*. Velasco et al. [70] harden the Linux kernel by implementing a triplication technique for the Mutex semaphores.

1.4.2 Error Correcting Codes

Error correcting codes work by appending a checksum to all stored values, which is calculated in a specific way from the individual bits of the value. The checksum contains redundancy information that enables detection of bit-flips and makes it possible to recover the original value. Of course, the number of bit-flips that an error correcting code can detect and repair is limited. Typically, it depends on the checksum length.

The major disadvantage of error correcting codes is the storage and performance overhead. The former occurs because the checksum is appended to the value, thereby increasing its size. The latter occurs because the checksum needs to be recalculated whenever the value is modified. Also, the increased value size can lead to worse cache performance, since fewer values fit into the cache. Of course, verifying the checksum and recovering from bit-flips has some performance overhead too.

There are various variants of error correcting codes with different tradeoffs between performance overhead and robustness. Some popular examples are:

- **Parity Bits:** Parity bits are single bits that are added to the end of a binary sequence so that the total sum of 1-bits is even or odd. They can only detect an odd number of bit flips because the bit-sum is identical for an even number of bit flips. Error recovery is not possible since the position of the flipped bit is unknown. Parity bits are very cheap to compute and often used in hardware, for example in PCI buses or in CPU instruction caches.

- **Hamming Codes:** Hamming codes [71] add a multi-bit checksum to values, with a length that depends on the size of the value. They allow detection of up to two bit-flips and recovery from single bit-flips. Hamming codes are commonly used at hardware level in ECC memory. Software implementations of hamming codes often incur a high performance overhead since the code is calculated over individual bits, which has a high overhead because the main memory is only byte-addressable.

- **Reed-Solomon Codes:** Reed-Solomon codes [72] are very flexible codes with a wide range of applications. Depending on the chosen checksum size `t`, the code can detect up to `t` bit-flips (without correction) or can correct up to `t/2` bit-flips. Reed-Solomon codes are for example used for CDs, QR codes, and DSL data transmission.

Error correcting codes cannot guarantee the absence of bit-flips, but they can improve the robustness against such soft errors and provide a reasonable level of certainty that a value is bit-wise consistent.

1.5 NVRAM Frameworks

With the rise of NVRAM, several frameworks and libraries are developed for managing NVRAM. There are two primary approaches: One approach is to treat NVRAM like main memory and directly used like normal DRAM. This is the approach that our framework uses. The alternative approach is to treat DRAM as a storage device and introduce it as an additional layer between main memory and secondary memory. An example is the Intel Optane technology that supplements an SSD with NVRAM to improve its performance.

The challenge of using NVRAM directly as main memory is the decreased reliability. While DRAM is highly reliable and only affected by rare radiation-induced bit-flips, write operations to NVRAM can often lead to inconsistent stored data because of technical limitations described in Section 1.3.2.

Martens et al. [32] present a library for persistent memory management that combines software transactional memory with error correcting codes to implement reliable transactions. It provides a `malloc`-like interface that can be used from both userspace and kernel code. To improve performance, they offload write operations to volatile RAM and use early detection of power failures to achieve durability on demand.

While our solution takes a similar approach, it differs in many fundamental aspects. Most notably, we use the type system of the Rust programming language to provide additional compile time guarantees and rule out programmer errors. For example, our framework detects missing checksum protections for data stored in NVRAM and it is able to completely prevent pointer consistency violations because of partial restores. Further, we provide an attribute to make static variables persistent, instead of only providing a `malloc`-like interface.

Another difference is that, instead of using software transactional memory, we rely on the transactional properties of a *sum-and-copy* error correcting code. On one hand, this approach reduces reliability and makes error correcting codes non-transparent so that they need to be explicitly specified in the source code. On the other hand, it simplifies the implementation since no STM transactions are needed and it considerably improves performance since it avoids the potentially high overhead of STM. It also allows the programmer to manually optimize checksum recalculations.

Chapter 2

Background

In this chapter we provide an introduction to the technologies used in our work. We use the *Rust programming language* to implement our NVRAM framework and make extensive use of its type system in order to provide additional compile-time guarantees. Section 2.1 gives an overview of the language and presents a selected set of features that are relevant for our implementation.

For evaluating our solution, we integrate our framework into *Redox OS*, which is a microkernel operating system written in Rust. Section 2.2 introduces the system and gives a short overview of its design and implementation.

2.1 Rust

Rust is a memory-safe system programming language. Like C and C++, it has only a minimal runtime and no garbage collector, which makes the language usable in many environments, including operating system kernels. The main feature of Rust is its powerful ownership system that guarantees memory and thread safety at compile time without any runtime overhead. It also features a powerful type system, a package manager, and C++-like performance.

Our framework heavily relies on Rust to provide its extensive compile time guarantees. Rust's *ownership and borrowing system*, which we describe in Section 2.1.1, guarantees memory and thread safety and thus rules out dangling pointers and data races at compile time. Through the advanced *trait system*, we are able to limit the types that can be placed in NVRAM in order to guarantee pointer consistency after a restore. We describe the trait system and its interaction with generic programming and dynamic dispatch in Section 2.1.2.

Section 2.1.3 gives an overview of the *procedural macro* feature of Rust, which allows to provide custom attributes and syntactical additions to the language. We use this feature to implement a `#[restorable]` macro that makes a static variable restorable with minimal boilerplate.

In Section 2.1.4 we show how *uninitialized memory* can be safely used from Rust. This is relevant for our framework since corrupted memory and uninitialized memory are very similar from the compiler's perspective.

2.1.1 Ownership and Borrowing

The *ownership and borrowing* system is the central feature of Rust. It guarantees memory safety without requiring a garbage collector by strictly enforcing fundamental invariants on all code, namely:

- Every value has an **single owner** at all times. When the owner drops the value, for example by letting it go out of scope at the end of the function, the value is no longer needed and can be safely deallocated.
- An owner can give other entities temporary access to a value by **borrowing** it. A borrow only lasts a limited, statically-known time and a borrowed value must not be dropped or invalidated. For the duration of the borrow, the owner cannot access the value.
- There are two types of borrows: **shared** and **exclusive borrows**. An exclusive borrow gives write access, but only a single exclusive borrow must be active at any point in time. The syntax for an exclusive borrow is the reference operator `&mut`. Shared borrows only allow read-only access, but multiple shared borrows can be active at the same time. Shared borrows are created using the `&` reference operator.

By enforcing these rules, use-after-free vulnerabilities can be fully prevented. The reason is that only the single owner can drop a value and that it can only be dropped when no borrows are active. Additionally, the borrowing rules enforce that values can be only mutated when they are not shared. This way, shared mutable state is prevented at compile time, which rules out common classes of errors such as data races or iterator invalidations. These safety guarantees make Rust a good choice for safety-critical software such as operating system kernels.

Listing 2.1 shows an example for Rust's ownership system. In line 3, a new growable `String` instance is allocated on the heap. This instance lives as long as the surrounding block, which ends in line 5. At this point, the destructor of the `String` type runs, which frees the heap memory again. In Rust, blocks are also expressions that evaluate to last expression if it is not terminated by a semicolon. In line 4, we let the block evaluate to a reference to `x`, thereby creating a shared borrow. The reference is then stored in the binding `s`. In line 6, we use the `println!` macro to output the value of `s` on the console.

```
1 fn main() {
2     let s = {
3         let x = String::from("hello");
4         &x
5     };
6     println!("{}", s);
7 }
```

Listing 2.1: Rust Ownership Example

```
error[E0597]: `x` does not live long enough
--> src/main.rs:4:9
  |
2 |     let s = {
  |         - borrow later stored here
3 |         let x = String::from("hello");
4 |         &x
  |         ^^ borrowed value does not live long enough
5 |     };
  |     - `x` dropped here while still borrowed
```

Listing 2.2: Compiler Output for Rust Ownership Example

The problem in this code example is that a reference to the `String` instance is used after it is deallocated in line 5. If the compiler would permit this code, an use-after-free error would occur. By keeping track of the *lifetime* of all values at compile time, the Rust compiler can detect violations of the borrowing rules. For the code in Listing 2.1, the compiler correctly detects the mismatch of lifetimes, which indicates that the string is dropped before the borrow ends. For this reason, it outputs the error shown in Listing 2.2.

2.1.2 Traits

Rust uses *traits* to define shared behavior with other types, comparable to *interfaces* in other languages. Listing 2.3 shown the definition of an example `Area` trait that can be implemented by structs representing geometric shapes. Listing 2.4 shows an example implementation of the trait for a `Square` struct.

```
pub trait Area {
    fn area(&self) -> f32;
}
```

Listing 2.3: Example Trait Definition

```
pub struct Square { width: f32, }

impl Area for Square {
    fn area(&self) -> f32 {
        self.width * self.width
    }
}
```

Listing 2.4: Trait Implementation for a Square Struct

```
pub fn print_area<A: Area>(shape: &A) {
    println!("{}", shape.area());
}
```

Listing 2.5: Example Function Generic over the Area Trait

Listing 2.5 shows a example function that is generic over the `Area` trait. The function can be invoked with any type that implements the trait, including the `Square` struct. Rust *monomorphizes* generic function, i.e. it creates a separate copy of the function for each type that it is used with. This avoids the overhead of dynamic dispatch and allows the compiler to perform more optimizations.

Dynamic Dispatch

When dynamic dispatch is desired, Rust allows to use *trait objects* to create non-generic functions. Trait objects are types with a *vtable* that makes it possible to invoke methods on the type without knowing its layout. Listing 2.6 shows a non-generic implementation of the `print_area` function that takes a reference to a trait object through the `dyn Area` syntax. This function is not monomorphized by the compiler, so it exists only in a single variant. The `shape.area()` call looks up the `area` method in the vtable, like it is done in object oriented languages such as Java.

```
pub fn print_area(shape: &dyn Area) {
    println!("{}", shape.area());
}
```

Listing 2.6: A Non-generic Function that uses Dynamic Dispatch

```
let sql = sql!(SELECT name FROM users WHERE id=1);
```

Listing 2.7: A Procedural `sql!` Macro

Auto Traits

A special kind of traits are *auto traits*. Auto traits are purely marker traits, which means that they do not define any methods or associated constants. Instead, they only convey the Boolean information whether a type fulfills the represented property.

Auto traits are automatically implemented for compound types like structs or enums if all their fields or variants implement the trait. Types can override this automatic behavior by providing an explicit positive or negative trait implementation. Negative trait implementations remove a trait implementation for a type through the `impl !Trait for Type {}` syntax. They are currently only available for auto traits, not for normal traits.

An example for an auto trait from Rust's standard library is the `Send` trait that is implemented for all types that can be safely transferred to other threads. This trait is implemented for most types, but there are important exceptions such as the `Rc` wrapper type for reference counted values. Since `Rc` uses a non-atomic integer for storing the reference count, there would be data races if the reference count is modified concurrently in different threads. For this reason, `Rc` explicitly opts-out of `Send` by providing a `!Send` implementation. This also means that all compound types that contain a `Rc` type are transitively not `Send` either.

2.1.3 Procedural Macros

Procedural macros are a powerful feature of Rust that allows arbitrary syntactical modifications. Rust currently supports three different types of procedural macros [73]:

- **Function-like Macros**: These macros can be used like functions and are useful for performing compile time checks and transformations. For example, it is possible to define a `sql!` macro for constructing a SQL query, as shown in Listing 2.7. The macro parses the query statement, checks it for syntactical errors, and then converts it to an internal SQL query type. This all happens at compile time, so that syntax errors at runtime can be avoided.

- **Attribute-like Macros**: This type of macro allows to create new attributes that can perform arbitrary modifications on the attributed item. For example, a serialization framework could define a `#[json(name = "")]` attribute for struct fields to specify the field name after serialization.

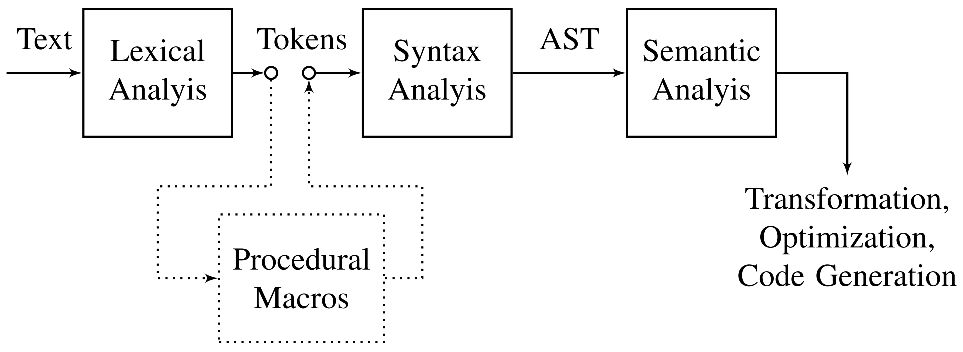

Figure 2.1: Compiler Pipeline with Procedural Macros

- **Derive Macros**: These macros allow the automatic implementation of a trait through a `#[derive(TraitName)]` attribute on `struct` and `enum` definitions. This attribute cannot modify the item definition, but they can provide additional trait implementations. For example, the `Clone` trait could be derived by providing an implementation that clones all fields of a structure.

Figure 2.1 shows how procedural macros are integrated into the compiler pipeline. First, the text of a source file is tokenized in the *lexical analysis* stage, thereby turning individual characters into tokens such as `<while>` or `<struct>`. Normally, the tokens are then passed to the *syntax analysis* stage, which verifies that the tokens describe a syntactically correct program and builds an abstract syntax tree (AST). This syntax tree is then passed to the *semantic analysis* stage, that performs type checking, object binding, and various other checks. If the program passes this stage, a semantically attributed syntax tree is then passed to the following stages, which transform the tree into an intermediate language, perform optimizations, and finally create code for the target platform.

As shown in the figure, procedural macros intercept the token stream between the lexical and syntax analysis stages. They directly work on the token stream and can perform arbitrary modifications, which even allows custom syntax extensions. However, they do not have access to any type or object binding information since they are invoked before the semantic analysis stage.

Since working directly on a token stream is not practical, Rust provides the `syn` library to parse the token stream into `struct` definitions. To turn the potentially modified parsed values back into tokens, the `quote` library can be used. This library also supports creating arbitrary additional Rust code.

2.1.4 Uninitialized Memory

Like most compiled languages, Rust assumes a well defined memory layout for values. For example, a Boolean value can either have the binary value `0` (representing `true`) or `1` (representing `false`). Other values (e.g. `3`) are prevented by the type system, which allows the compiler to perform certain optimizations. This leads to problems when working with corrupted or uninitialized memory because unexpected bit-representations of types can lead to invalid optimizations and undefined behavior.

To support safe usage of undefined memory, the Rust standard library defines the `MaybeUninit` wrapper type. The type makes it possible to safely work with uninitialized memory by preventing the compiler from making any assumptions about the wrapped value. This is also useful when working with potentially corrupted memory: Only after the consistency of the memory was verified, the actual value is constructed from it.

`MaybeUninit` is implemented as a C-style `union` of the wrapped type and the empty tuple type `()`. This way, the compiler cannot make any assumptions about the wrapped memory because it does not know which variant is stored in the `union`.

2.2 Redox OS

To evaluate our work, we apply our framework to the `Redox` operating system. Redox OS is an Unix-like microkernel operating system written in Rust. Like Rust itself, it is dual licensed under the MIT/Apache2 licenses and developed as an open-source project. While Redox is not self hosting yet, it already runs a rudimentary web browser and supports many traditional Unix programs.

In contrast to the *everything is a file* principle of Unix systems, Redox employs an *everything is an URL* principle. The idea is to prepend paths with a so-called *scheme* specifier such as `file:`, similar to the protocol specifier in URLs. Through different schemes, different resources can be accessed, including traditional files (`file:`), networking sockets (`tcp:` and `udp:`), and graphical output (`display:`). As an effect, each resource type has its own namespace, instead of sharing a single namespace as on Unix systems.

Each scheme has an associated driver, either provided by the kernel or by an userspace program. Programs can dynamically register a new scheme through the *root scheme*, which is denoted by a single colon (`:`). This makes it possible to use schemes as the main form of inter-process communication.

2.2.1 Components

Figure 2.2 gives an overview of the components of Redox OS, including a subset of the available schemes. Redox is a microkernel operating system, so most functionality and drivers are provided by userspace programs. The kernel provides the following main components:

- The **system call handler** allows userspace programs to invoke kernel functionality and communicate with other processes such as drivers. Its main tasks are the validation of incoming system calls, to ensure that the program is eligible to perform the specified operation, and the subsequent invocation of the responsible kernel component. For example, it dispatches the `fork` system call to the *context manager* and the `open` system call to the *scheme manager*.

- The **context manager** is responsible for managing threads and processes, which are called *contexts* in Redox. It provides functionality to create new contexts, keeps track of the state and resources of all existing contexts, and provides a scheduler to decide which context is run next.

- The **memory manager** manages both the physical and virtual memory. It allocates additional memory to programs if requested and also supports the allocation of continuous physical memory, for example for *direct memory access* (DMA) operations. Further, it provides a kernel-internal heap memory region and a corresponding allocator.

- The **scheme manager** keeps track of all available schemes. It dispatches each scheme operation to the responsible driver, which can either live in kernel space or userspace.

Additionally, the kernel provides drivers for several kernel-level schemes. The root scheme (`:`) allows the creation of additional schemes by (userspace) programs. The `debug:` scheme makes it possible to write debug output from inside the kernel, for example through a serial port. The `irq:` scheme provides access to hardware interrupts, which is important for device drivers. There are also some kernel-internal schemes, for example for implementing the *pipe* communication primitive.

The arrows in Figure 2.2 give an overview how system calls are dispatched. When a userspace process issues a system call, the system call handler is invoked. After validating the request, it is passed to the responsible kernel component, such as the context or memory manager. For scheme operations, the request is passed to the scheme manager, which contains an internal mapping of scheme specifiers to the corresponding drivers. These drivers can live either in the kernel or in

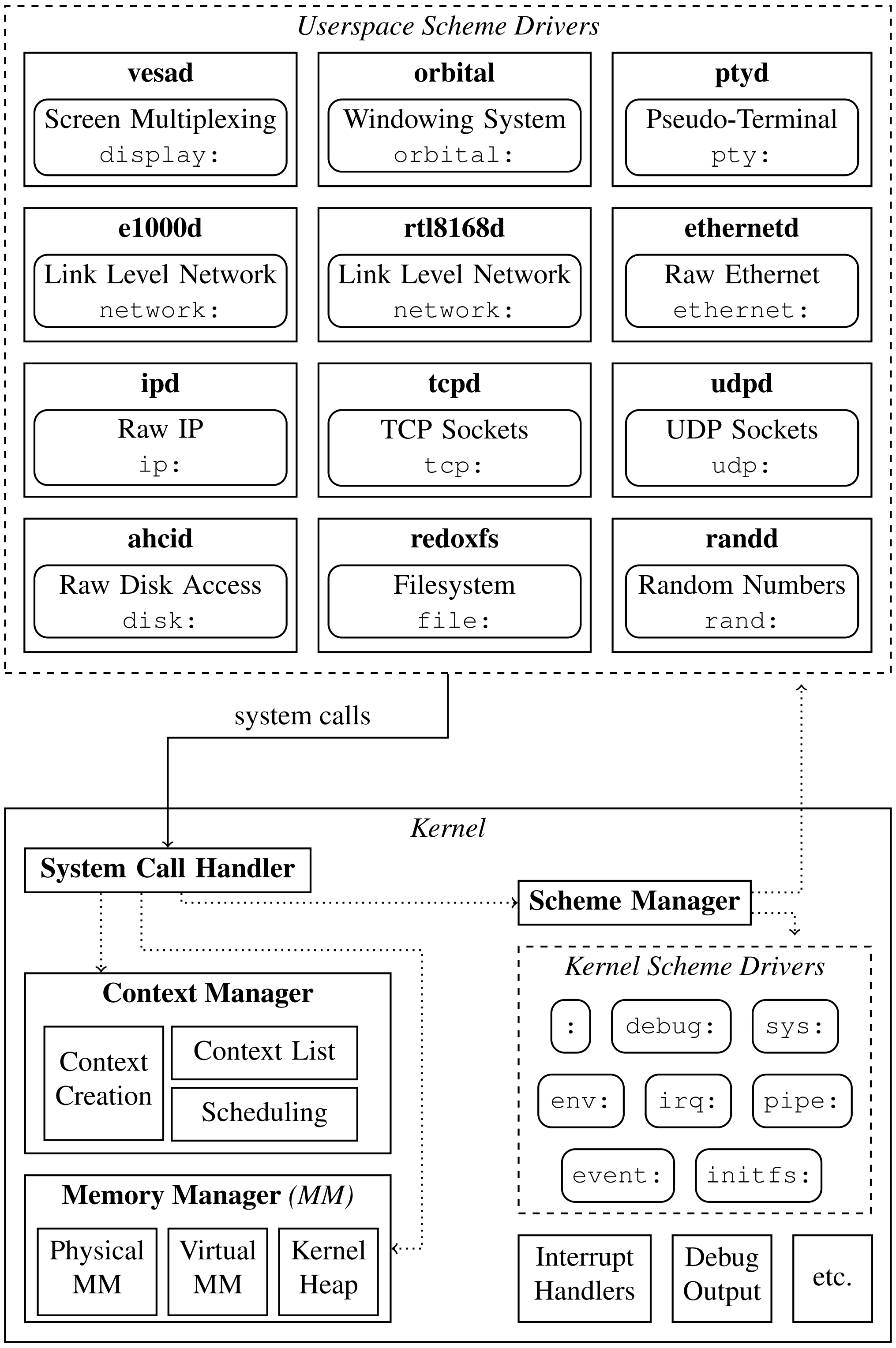

Figure 2.2: Userspace and Kernel Components of Redox OS

userspace. After the driver performed the requested operation, it returns a result to the scheme manager, which is then passed through the system call handler back to the process that invoked the system call.

2.2.2 Boot Process

The boot process of Redox OS starts with the bootloader, which performs some fundamental CPU initializations and loads the kernel. Then control is transferred to the entry point of the kernel. The kernel first performs platform specific initialization routines to set up components such as the memory management unit and the interrupt handler. On `x86_64`, it creates a physical frame allocator, initializes a 4-level page table, and creates global and interrupt descriptor tables. Then it creates a kernel heap, sets up graphical debug output, and initializes fundamental hardware devices such as the interrupt controller and the *high precision event timer*. Afterwards, it transfers control to the platform independent start function of the kernel, which is called `kmain`.

The `kmain` function of the kernel first initializes the context manager and creates an initial root process. The other kernel components initialize themselves lazily when they are used the first time, so the `kmain` function can directly initialize the userspace afterwards. For that, it loads an userspace initialization program from the initialization filesystem through the `initfs:` scheme. Then, it creates a new context for the program and transfers control to it.

Instead of hardcoding an initialization sequence, the userspace initialization program reads and interprets an `/etc/init.rc` script from the `initfs:` scheme. For a normal desktop configuration, this script initializes the display through the `vesad` program, calls the `pci` program for initializing PCI devices, and then launches `redoxfs` to create the initial filesystem. Afterwards, it executes all second-level initialization scripts in the `/etc/init.d` directory of the `initfs:` scheme. Depending on the configuration, these scripts initialize other userspace programs such as the login shell, the network daemons, the window manager, and pseudo-terminals.

By specifying the userspace initialization routines in separate scripts, it is easily possible to configure the system for different use cases. For example, by supplying different initialization scripts, Redox OS can be configured as a desktop system with a graphical user interface, as a server system with only a console-based interface, or even as a minimal embedded system.

Chapter 3

Design

The idea of our NVRAM framework is to map the NVRAM as a second, persistent heap into programs. In contrast to the default heap that is reinitialized on each reboot, the persistent heap keeps its content across system restarts. Through a *persistent RAM manager*, programs can allocate chunks of NVRAM for persisting selected state. The design of the persistent RAM manager is described in Section 3.1.

Treating the NVRAM as heap memory instead of as a storage device allows programs to directly operate on it. This makes it possible to persist more state of the program, instead of only persisting parts of the file system. The drawback of this approach is that NVRAM has reliability issues, as noted in Section 1.3.2. Therefore, our framework protects data on the restored heap with error correcting codes. Section 3.2 describes these error correcting codes and explains how Rust's type system can be used to enforce data protection.

In contrast to the *whole system consistency* approach proposed by Narayanan et al. [26] that persists the complete heap memory, our framework uses the NVRAM as a second, separate heap region and requires explicit action from programs for making use of it. We decided for this approach because it allows the gradual transformation of a system. For example, it makes it possible to implement support for file system persistence without requiring any code changes in other components. In contrast, whole system persistence requires extensive code modifications, including adjusting all device drivers to correctly reinitialize hardware devices to their previous state after a restart.

The problem with persisting only a part of the heap state is that it leads to partially restored state after a restart. As noted in Section 1.3.3, partial restores can lead to violations of both pointer and semantic consistency. For this reason, our framework utilizes the type system of Rust to limit which types are allowed to be stored in NVRAM. Section 3.3 shows how our framework can prevent invalid partial restores and guarantee pointer consistency using this approach. In

Section 3.4, we then summarize the consistency guarantees of our framework.

Since global state is often stored in static variables, our framework provides a `#[restorable]` attribute that makes a static variable persistent with minimal code changes. Section 3.5 explains how the attribute persists static variables by implicitly storing them on the restorable heap instead of keeping them in the memory of the executable.

By combining these individual components, we get a framework that allows the safe and ergonomic usage of NVRAM from both operating system kernels and userspace programs.

3.1 Persistent RAM Manager

The central component of our framework is the `PersistentRamManager`. It is responsible for allocating NVRAM to programs and managing existing allocations across reboots. Each program instance has an own instance of the `PersistentRamManager` to strictly isolate allocated NVRAM from other processes. Similar to normal heap allocators that query memory from the kernel using a system call such as `sbrk` on Linux, the `PersistentRamManager` queries slices of NVRAM from the kernel through a custom system call.

The interface of the manager provides two kinds of allocations: named and unnamed allocations. Named allocations can be used to recreate program state after reboots. Section 3.1.1 describes named allocations and outlines the problem of selecting unique keys. Unnamed allocations are similar to normal heap allocations, with the difference that they are allocated on the persistent heap. They are described in Section 3.1.2. In Section 3.1.3, we then present an example for named and unnamed allocations that shows the different use cases for the two allocation types.

3.1.1 Named Allocations

Named allocations are identified by an unique key. When an allocation with the same key is queried again (e.g. after a system restart), the persistent RAM manager returns a pointer to the existing allocation instead of creating a new one. This makes it possible to restore state when a program is launched again.

Special precaution must be taken when choosing a key, as it must be unique per program instance. If two named allocations inside the same program instance use the same key, the wrong allocation might be used when the program is resumed after a reboot. However, the key of each allocation must remain the same across reboots because otherwise the previous allocation is not found.

For this reason the manual creation of named allocations is not recommended. Instead, our framework implements support for persistent static variables through the `#[restorable]` attribute (described in Section 3.5). Using this attribute, no key collisions can occur.

Deallocation of named allocations is supported through a `deallocate_named` method. Our framework does not provide abstraction types that automatically handle deallocation yet, so the programmer has to manually invoke the deallocation function. Future work might add a type similar to Rust's `Box` type that automatically calls the deallocation function when it goes out of scope.

3.1.2 Unnamed Allocations

Unnamed allocations always return a new, unused chunk of NVRAM. While the persistent RAM manager keeps track of named allocations in order to let programs find their previous state, it keeps no data about unnamed allocations. To reuse an unnamed allocation after a system restart, the program needs to keep a pointer to the allocation. For this reason, unnamed allocations are mostly useful for "inner" allocations, i.e. allocations that are stored inside another (named) allocation.

Since unnamed allocations always return a new chunk of NVRAM, they should only be created when initializing other persistent data. Programs that unconditionally create new unnamed allocations on every reboot can lead to memory leaks and out-of-memory errors.

While the manual deallocation of unnamed allocations is possible, we recommend to use unnamed allocations together with the allocation types of the standard library. By specifying a `PersistentRamAllocator` provided by our framework as a second generic parameter to allocation types such as `Box`, the types implicitly perform unnamed allocations from the `PersistentRamManager` instead of using the default heap. The types automatically handle their own deallocation when they go out of scope, thereby preventing programmer errors.

3.1.3 Allocation Example

Listing 3.1 shows an example usage of named and unnamed allocations. The `Process` struct contains an ID and a heap allocated pointer (using the `Box` type) to a `ProcessData` struct (not shown). The purpose of the `new_persistent_process` function is to create a new process instance on the persistent heap. For that, it queries a *named* allocation from the persistent RAM manager. If a previous allocations exists (i.e. after a reboot), the manager returns a pointer to it. Otherwise, it creates a new allocation and initializes it with the given closure, which invokes the `new_process` function when called.

```
struct Process {
    id: u64,
    data: Box<ProcessData>,
}

fn new_persistent_process(name: &str) -> Box<Process> {
    let key = get_unique_key_for_process(name);
    PERSISTENT_RAM_MANAGER.allocate_named(key, || {
        new_process(name)
    })
}

fn new_process() -> Process {
    let data = ProcessData::new();
    Process {
        id: next_id(),
        data: PERSISTENT_RAM_MANAGER.allocate_unnamed(data)
    }
}
```

Listing 3.1: Pseudo-Code Example for Named and Unnamed Allocations

The `new_process` function first creates a new `ProcessData` instance and allocates it on the restored heap as an *unnamed* allocation. Then, it creates a `Process` struct with the data instance and returns it. The reason that the function uses an unnamed allocation is because the address of the allocation is stored inside a persistent `Process` instance. By recovering the previous value of the `Process` instance from the persistent RAM manager after a restart, the pointer to the unnamed allocation is recovered too.

The code for the `get_unique_key_for_process` function is not shown here. Its purpose is to create an unique integer key from the given (unique) process name. As noted in Section 3.1.1, the key must be unique per process instance and remain the same across restarts. Since a non-unique key can lead to restoration of the wrong state on a system restart and cause undefined behavior, it is not recommended to create the key manually. Instead, a static variable annotated with the `#[restorable]` attribute should be used to safely create named allocations.

3.2 Ensuring Bit-wise Consistency

Since NVRAM is susceptible to bit-flips (see Section 1.3.2), the memory representing a value might no longer be valid after a system restart. This can lead to errors and vulnerabilities because restoring a value with an invalid representation can cause undefined behavior. For example, the Rust compiler assumes that a `bool` can only have the values `0` and `1`, representing `true` and `false`. If a `bool` has a different value (e.g. `3`) after a restart, undefined behavior occurs.

To avoid undefined behavior, our framework needs to ensure that no bits are flipped when restoring a value. Since bit-flips occur randomly at runtime, there is no way to completely prevent them, therefore it needs to detect and recover from them. A common way to solve this problem is to employ *error correcting codes* (ECCs), as introduced in Section 1.4.2.

Section 3.2.1 discusses the tradeoffs of different ECC variants and presents the variant that we use for our framework. Section 3.2.2 then explains our reasons for integrating the chosen ECC variant as an explicit component, instead of adding it to values implicitly. Finally, Section 3.2.3 shows how our framework employs Rust's type system to ensure that all restorable data is protected by an ECC.

3.2.1 ECC Design

As noted in Section 1.4.2, error correcting codes (ECCs) add redundancy to a value by appending some kind of checksum. Dependent on the used ECC, these checksums can be used to detect and correct a certain number of bit-flips. Since the checksums need to be recalculated whenever the value is modified, they decrease write performance. Read performance is also decreased because the checksum is regularly verified, often on every access. Further, ECCs increase the size of the value, which can lead to worse cache behavior.

To limit the performance overhead as much as possible, there are various ECC designs with different use cases and tradeoffs. For example, if only single bit-flips should be detected, parity bits can be used, which are very cheap to compute. For our framework, we require an error correcting code that can be efficiently implemented in software and is able to protect heap variables from (multiple) bit-flips.

Borchert et al. [64] compared the suitability of different ECC designs for use in kernel heap data structures. The performance evaluation showed that complex codes that operate on bit-level such as Hamming codes have a much higher performance overhead than simpler codes that rely on copies of the whole value. They found that an ECC design based on a 32-bit word sum and a copy of the value is most favorable for protecting heap memory against radiation induced bit-flips. Since the use case is similar, we also use this *sum-and-copy* ECC design for our framework.

Instead of calculating a checksum over individual bit values, the sum-and-copy ECC uses a sum of the 32-bit words of the value and a complete copy of the value as checksum, as shown in Figure 3.1. The sum is calculated by splitting the value into 32-bit chunks and adding these chunks together using wrapping addition. The

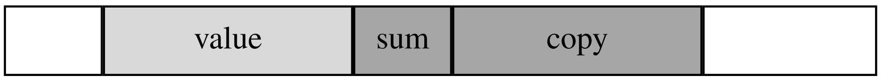

Figure 3.1: The Memory Layout of a Value with a Sum-And-Copy ECC

copy is just a bit-wise copy of the value. Both operations are very fast because they operate on native CPU word sizes and linearly traverse memory.

The consistency of a protected value is checked by calculating the sum again and comparing it with the sum stored in the checksum. If they match, it is likely that value still has its original value. Else, the copy that is part of the checksum is checked for consistency by comparing its sum too. If the copy is consistent, the original value can be restored by overwriting it with the copy. If the copy is inconsistent too, the restoration fails.

While this approach is inferior to bit-based error correcting codes such as Hamming [71] or Reed-Solomon [72] codes from a theoretical perspective, it works better for protecting the heap against random bit-flips because of two reasons. First, it can be computed very efficiently because both the sum and the copy are very simple operations that are very cheap on the CPU and main memory. Second, bit-flips occur relatively seldom so that a theoretical inferior approach suffices in practice. Namely, it is improbable that both the value and the copy are affected by bit-flips in the same time interval, so that a cheap-to-compute checksum is still robust enough.

In addition to protection against soft errors because of unreliable memory (see Section 1.3.2), the sum-and-copy ECC also provides limited protection against partial writes (see Section 1.3.3). The reason is that the `copy` and `sum` fields are only updated after the value was fully written. When a partial write occurs because of a sudden system restart, the violated bit-wise consistency of the value is correctly detected because the stored `sum` field no longer matches. Provided that no bit-flip has occurred, a restoration of the previous value is possible through the `copy` field.

3.2.2 Explicit and Implicit ECCs

Error correcting codes (ECCs) can either be applied explicitly or implicitly. Explicit ECCs are directly specified in the program code and thus give the programmer maximal control. Implicit ECCs are transparently added, for example by a precompiler, and consequently require no code modifications.

Borchert et al. [64] apply implicit ECCs to data by utilizing *aspect-oriented programming* [74]. Aspect-oriented programming (AOP) allows to let a program perform custom actions on specified conditions. The authors use AOP to implement a generic ECC protection for C++ objects. The idea is that each object method is augmented with an ECC check at the beginning and an ECC recalculation at the end. This way, no code modifications are necessary.

The disadvantage of implicit ECCs is that they take away control from the programmer, which might result in suboptimal performance. For example, with explicit ECCs the programmer could find more optimal places for ECC recalculations instead of performing them after every, possible very small, method call. Another disadvantage is that most languages do not have direct support for AOP, so that a special precompiler is needed. For example, Borchert et al. [64] require the *AspectC++* compiler, which translates aspect oriented source code into standard C++ code.

For our framework, we decided to use explicit ECCs in order to give the programmer full control. Another factor that influenced our decision is that no mature tooling for aspect-oriented programming in Rust exists yet, so we would have to create our own solution. Furthermore, the Rust programming language generally favors explicitness, so an explicit ECC implementation fits the language well.

To reduce the effects of the most major drawback of explicit ECCs, our framework provides special wrapper types that can be used with only minimal code modifications. The idea is to augment common synchronization wrapper types such as `Mutex` or `RwLock` with an ECC, while keeping the original API unchanged. For example, the `EccMutex` type keeps an additional ECC for the wrapped value, which is checked when the value is locked and recalculated when the lock is released again. This way, no code modifications apart from type renames are required.

3.2.3 Requiring ECCs for Restorable Values

To ensure bit-wise consistency of restored state, an error correcting code must be applied to each restorable value. This can be easily forgotten in complex code bases, for example in nested compound types that refer to other restorable values through pointers. To prevent programmer errors, our framework uses Rust's type system to guarantee that every restorable value is protected by an error correcting code.

As mentioned in Section 2.1.2, Rust uses a concept called *traits* to define shared behavior of types, similar to interfaces in languages like Java. Our Framework defines a trait called `ConsistencyCheckable` that describes values that can check their own consistency, for example by an internal ECC checksum. The definition of the trait is shown in detail in Section 4.3.

By requiring the `ConsistencyCheckable` trait for all values that are allocated from the persistent RAM manager, the compiler verifies at compile time that each value is protected by an error correcting code. Listing 3.2 shows an example

```
struct Item {
    data: u64,
}

fn new_persistent_item() -> Box<Item> {
    let item = Item { data: 0 };
    PERSISTENT_RAM_MANAGER.allocate_unnamed(item)
}
```

```
error[E0277]: the trait bound `Item: ConsistencyCheckable` is
not satisfied
 --> src/main.rs:7:30
  |
7 |     PERSISTENT_RAM_MANAGER.allocate_unnamed(item)
  |                            ^^^^^^^^^^^^^^^^ the trait
`ConsistencyCheckable` is not implemented for `Item`
```

Listing 3.2: Attempt to Allocate an Item without ECC Protection (in Pseudo-Code) and the Resulting Error Message

function that attempts to perform an unnamed allocation for an `Item` type that has no ECC protection. The type does not implement the `ConsistencyCheckable` trait, therefore the shown compilation error occurs. To fix this example, the `Item` type could be wrapped into the `Ecc` wrapper type presented in Section 4.5.1.

3.3 Guaranteeing Pointer Consistency

While error correcting codes provide reasonable safety that the bit-representation of values remains unchanged after a restart, bit-wise consistency does not guarantee that pointers are still valid. For example, a pointer that points to a value on the normal heap becomes invalid after a restart because the pointed value no longer exists. Such an use-after-free vulnerability is a common cause of exploits and thus must be prevented.

Our framework solves this problem by permitting only certain pointer types in restorable values. Namely, only pointers that are guaranteed to point to the restored heap are permitted. This limitation is ensured by requiring a custom *auto trait* called `RestoreSafe` for all restorable values. Section 2.1.2 described how auto traits are automatically implemented for compound types such as structs if all their fields implement the trait. It also explained that it is possible to override the derived implementations by providing explicit positive or negative implementations for a type.

The `RestoreSafe` auto trait works by treating all pointer and reference types as potentially unsafe through a negative implementation. Since all collection types use pointers internally, they are not `RestoreSafe` either because of the transitivity of the auto trait mechanism. The result is that the trait is implemented exactly for types without any internal pointer or reference types. Additionally, we provide positive implementations for collection types that are guaranteed to be stored in NVRAM. The details of these implementations are described in Section 4.4.

By requiring the `RestoreSafe` trait for all restorable values, the compiler guarantees that no violations of pointer consistency can occur when restoring state after a system restart. Since this check is performed at compile time, no runtime overhead occurs.

3.4 Summary of Consistency Guarantees

This section summarizes how the design of our framework prevents or at least considerably reduces most types of consistency violations that were mentioned in Section 1.3.

Unreliable memory can lead to random bit-flips at runtime. While it is not possible to fully prevent such soft-errors, our framework employs error correcting codes for all values stored in the particularly susceptible NVRAM. This way, most soft errors can be repaired or at least detected.

Partial restores cannot lead to dangling pointers in our framework because it employs Rust's type system to only permit types with safe pointers in NVRAM. Namely, it forbids storing types that contain pointers to non-restored data. While we do not use software transactional memory, our framework uses a copy-and-sum ECC that prevents *partial writes* to a limited degree.

Other causes for *dangling pointers* are ruled out by Rust's memory safety or inhibited by the use of error correcting codes. *Data races* are also prevented at compile time by the Rust programming language.

3.5 The `#[restorable]` Attribute

The purpose of our framework is to make parts of the state of a program restorable. For heap variables, our framework provides named and unnamed allocation methods to place the variable in NVRAM. However, global state is often also stored in static variables. For example, the Redox kernel stores the list of available *schemes* (see Section 2.2) in a static variable. As static variables are initialized at compile time and stored in the executable itself, it needs some tricks and boilerplate code to make a static variable restorable.

```
#[restorable(restore = true)]
static TEST_VALUE: Mutex<Ecc<u64>> = Mutex::new(Ecc::new(0));
```

Listing 3.3: Example Use of the `#[restorable]` Attribute

To make persisting static variables as convenient as possible, our framework provides a `#[restorable]` attribute that makes a static variable restorable with minimal boilerplate. The attribute works by lazily initializing the variable in the NVRAM on the first use, by creating a named allocation using the persistent RAM manager.

Listing 3.3 shows an usage example of the `#[restorable]` attribute. The attribute takes an argument named `restore` that controls whether the value should be restored after a reboot or newly initialized. No other code needs to be modified when the `restore` argument is switched between `true` and `false`. It is worth noting that the value is wrapped in the `Ecc` wrapper type (presented in Section 4.5.1), which provides the required error correcting codes for checking bit-wise consistency. It is also wrapped in a `Mutex` to allow safe concurrent mutation.

While the attribute works in a straightforward way for static variables of simple types such as `u64`, its usage becomes more complex for types with internal heap allocations. The reason is that these allocations should be placed either in NVRAM or on the normal heap depending on the value of the `restore` argument. Section 3.5.1 shows how this problem applies to the collection types of the standard library. Section 3.5.2 then explains how the `#[restorable]` attribute solves this problem by creating a special type alias for each static variable.

3.5.1 Restorable Collection Types

As we showed in Section 3.3, restoring types with internal pointers can lead to violations of pointer consistency if a restored value points to non-restored values. For that reason, the persistent RAM manager requires the `RestoreSafe` trait, which guarantees that type either contains no internal pointers, or only pointers that are guaranteed to live in the NVRAM.

An example for a pointer type that implements the `RestoreSafe` trait is `Box<u32, PersistentRamAllocator>`, which is a simple heap allocated value. The first generic parameter is the type of the pointed value, an unsigned 32-bit integer in this case. The second generic parameter is optional and specifies the heap allocator that should be used for allocating the needed backing memory. If the parameter is not specified, the normal heap is used by default. By specifying the `PersistentRamAllocator` as parameter, which allocates all memory from

NVRAM, it is guaranteed that the `u32` will be still valid after a reboot.

While it is possible to use this type in a static annotated with `#[restorable( restore = true)]`, it leads to unintended results for `restore = false`. The problem is that the value is newly allocated on each reboot, while leaking the previous value. On each reboot, it would fill the NVRAM further until no more memory is available. This problem also applies to the Redox OS kernel, since it makes extensive use of Rust's standard collection types internally. For example, it uses a `BTreeMap` to keep track of available *schemes* (see Section 2.2.1).

3.5.2 Allocator Type Aliases

To allow specifying types that only use the persistent RAM allocator if `restore = true`, the `#[restorable]` attribute creates a `_X_ALLOCATOR` type alias for each annotated static `X`. This type alias points to the persistent RAM allocator if `restore = true` and to the default global heap allocator if `restore = false`.

Listing 3.4 shows example code that uses the allocator type alias. The `Test` struct defines two fields, `value_1` and `value_2`. The `value_1` field is a simple `u32` integer. The `value_2` field is a heap allocated `u32`, which uses the `Box` type. Since the purpose of the `Test` type is to be stored in the static `TEST_VALUE` variable, it specifies the `_TEST_VALUE_ALLOCATOR` type alias as the second generic parameter to the `Box` type. This way, the allocation is performed in NVRAM when the `restore` argument is set to `true` and on the normal heap otherwise. Because of the second type parameter, the `Box::new_in` function must be used for initializing the `value_2` field, instead of the normal `Box::new` function that always allocates on the default heap.

The type alias is not a perfect solution since the programmer still needs to remember to use the type alias to prevent leaks, but it at least makes it possible to switch the `restore` argument for a static without requiring code modifications. Adding runtime or even compile time checks for leak prevention is left for future work. Note that memory leaks do not violate memory safety (i.e. no undefined behavior can occur), so this limitation of our framework does not diminish the safety guarantees summarized in Section 3.4.

```
struct Test {
    value_1: u32,
    value_2: Box<u32, _TEST_VALUE_ALLOCATOR>,
}

#[restorable(restore = true)]
static TEST_VALUE: Mutex<Ecc<Test>> = Mutex::new(Ecc::new(Test {
    value_1: 0,
    value_2: Box::new_in(1, _TEST_VALUE_ALLOCATOR),
}));
```

Listing 3.4: Example Use of the Allocator Type Alias

Chapter 4

Implementation

In this chapter, we outline the implementation of some key components of our NVRAM framework. Apart from describing the required kernel extensions and the implementation of the `PersistentRamManager`, we present the implementation details of the `RestoreSafe` and `ConsistencyCheckable` traits, which are responsible for some central consistency guarantees. We also describe the implementation of several ECC wrapper types and the `#[restorable]` attribute.

The chapter is structured as follows: Section 4.1 gives an overview of the kernel extensions that were required to add support for NVRAM to the system. Section 4.2 describes how the `PersistentRamManager` uses a modified system call to query a slice of NVRAM from the kernel. It also explains how the manager detects that previous state exists in NVRAM after a system restart. Section 4.3 then presents the definition of the `ConsistencyCheckable` trait, which is used by the persistent RAM manager to ensure that all values that are placed in NVRAM are protected by an error correcting code. In Section 4.4, we explain the implementation of the `RestoreSafe` trait, which relies heavily on Rust's type system. The trait is used to guarantee pointer consistency by limiting the pointer types that are allowed in NVRAM.

The implementation of error correcting codes is presented in Section 4.5. Our framework provides an `Ecc` wrapper type that adds a sum-and-copy ECC to an arbitrary type. It also implements a solution for checking the bit-wise consistency of nested types, i.e. types that contain internal pointers to other types. In order to reduce required code modifications, we further provide an ECC-augmented version of the common `RwLock` synchronization wrapper type.

In Section 4.6, we describe the implementation of the `#[restorable]` attribute. The attribute is implemented as a *procedural macro*, which is a feature of the Rust programming language that allows arbitrary source code transformations.

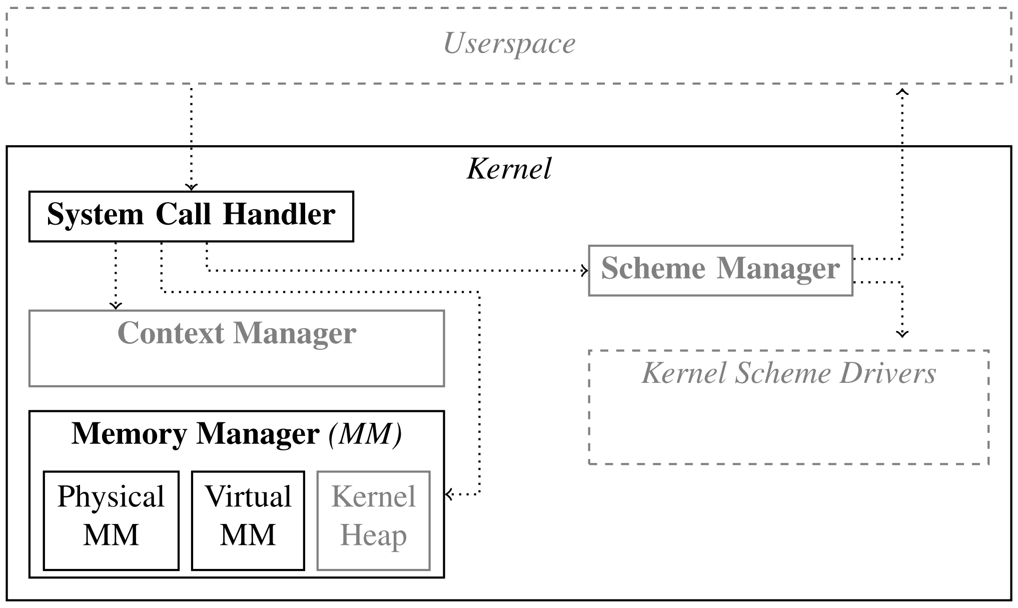

Figure 4.1: Modified Components of the Redox Kernel

4.1 Kernel Extensions

In order to add support for NVRAM, we modified two key components of the Redox kernel: the memory manager and the system call handler. Figure 4.1 gives an overview of the kernel structure again, with the unmodified components grayed out.

The memory manager was modified in the following way: First, we extended the physical memory manager to also manage available NVRAM in addition to available main memory. As we did not have access to real NVRAM, we simulated it by reserving a contiguous, physical main memory region (see Section 5.1.3). By reserving the region right after creating the physical memory map, we ensure that it remains the same across system restarts. Second, we added additional kernel-internal allocation methods to retrieve slices of NVRAM. Third, we extended the virtual memory manager with methods to map slices of NVRAM into the virtual address space.

In order to give userspace programs access to NVRAM too, we extended the system call handler. Instead of introducing a new system call type and increasing the API surface, we extended the existing `physmap` system call with an additional `PHYSMAP_PERSISTENT_RAM` flag. Normally, the call maps a given slice of physical memory at a free virtual address selected by the kernel. When the `PHYSMAP_PERSISTENT_RAM` flag is given, the kernel instead maps a newly allocated slice of NVRAM of the specified size.

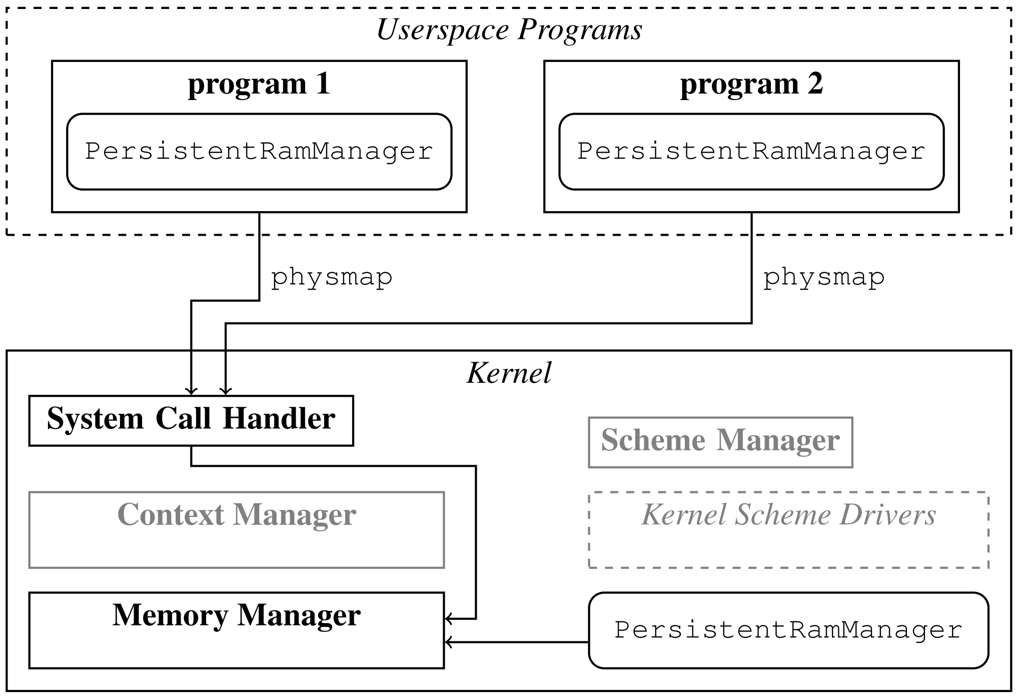

Figure 4.2: Integration of the `PersistentRamManager` into Redox

4.2 The `PersistentRamManager`

An mentioned in Section 3.1, the `PersistentRamManager` is the key component of our framework. It is responsible for managing NVRAM for processes and providing allocation methods. Through named allocations, it allows processes to retrieve existing state after system restarts. This functionality is also implicitly used by the `#[restorable]` attribute, which can be used for persisting static variables.

Figure 4.2 shows how the `PersistentRamManager` is added to processes. Each userspace process that uses NVRAM includes their own instance of the `PersistentRamManager`, which all operate on separate slices of NVRAM. To retrieve a slice of NVRAM, each instance invokes the modified `physmap` system call (described in the previous section) on initialization, which then retrieves a slice of NVRAM from the memory manager of the kernel. As shown in the figure, the `PersistentRamManager` can also be used from inside the kernel itself. Instead of using the `physmap` system call, it then queries the NVRAM slice directly from the memory manager.

To enable safe state restoration after a system restart, the kernel must ensure to allocate the same slice of NVRAM as before to each instance of the `PersistentRamManager`. One possible solution to this problem is to make pro-

```
pub unsafe trait ConsistencyCheckable {
    /// Check whether the given `Self` instance is valid,
    /// without changing it.
    fn is_consistent(self: *const Self) -> bool;

    /// Check whether the given `Self` instance is valid,
    /// potentially changing it.
    ///
    /// This method is allowed to fix the given `Self`
    /// instance, e.g. by applying error correcting codes.
    fn is_consistent_restore(self: *mut Self) -> bool;
}
```

Listing 4.1: Definition of the `ConsistencyCheckable` Trait

cess IDs persistent across restarts. This makes it possible to use the process ID as an unique key for retrieving the corresponding slice of NVRAM. Unfortunately, we did not manage to implement support for persistent process IDs in the Redox kernel yet.

As a straw man solution, we require a deterministic initialization order for all processes that use NVRAM for now. This allows the kernel to linearly hand out NVRAM slices without needing to keep track of the mapping between NVRAM slices and processes. For example, the filesystem driver is always started before the userspace test application (see Section 2.2.2), so that the filesystem driver always receives the first slice of NVRAM across restarts with a linear allocation strategy.

To detect whether previous state exists, each instance of the `PersistentRamManager` places a marker value such as `0x12121212` at the beginning of its NVRAM slice on initialization. By checking this value after a restart, the `PersistentRamManager` can detect that previous state exists in NVRAM and attempt a restoration. By overwriting the marker value, it is possible to force a reinitialization on the next reboot.

4.3 The `ConsistencyCheckable` Trait

The `ConsistencyCheckable` trait describes types that can check their own consistency. The trait definition is shown in Listing 4.1. Since the NVRAM framework relies on a correct implementation, the trait is defined as `unsafe` to signal that any implementation must satisfy certain invariants.

The trait defines two methods, `is_consistent` and `is_consistent_restore`. Both methods get a raw pointer to the `Self` instance as an argument

```
struct ParityBitProtected {
    value: u32,
    parity_bit: u8,
}

unsafe impl ConsistencyCheckable for ParityBitProtected {
    fn is_consistent(self: *const Self) -> bool {
        // convert the raw pointer into a reference
        let s = unsafe {&*self};
        // only consider the last bit
        let parity_bit = u32::from(s.parity_bit % 2);
        // verify parity to detect single flipped bits
        (s.value.count_ones() + parity_bit) % 2 == 0
    }

    fn is_consistent_restore(self: *mut Self) -> bool {
        Self::is_consistent(self)
    }
}
```

Listing 4.2: Example Implementation of the `ConsistencyCheckable` Trait

and return a Boolean value that signals whether the instance is bit-wise consistent. The difference between the two methods is that the latter is allowed to repair the `Self` instance if possible, while the former only has read-only access.

Passing the `Self` instance behind a raw pointer is an uncommon pattern in Rust. Normally, methods take either `&self` or `&mut self` as arguments, which desugars to `self: &Self` and `self: &mut Self` references. The reason for using raw pointers instead of references in the method definitions is that Rust has very strong guarantees about references, namely that they must always point to valid values. Since the `Self` value is potentially inconsistent, creating a reference to it could already invoke undefined behavior.

The simplicity of the method definitions makes it possible to implement the trait for a variety of error correcting code implementations. Listing 4.2 shows an example implementation for a `ParityBitProtected` struct that protects an `u32` value with a single parity bit. Instead of using a `bool` type for the `parity_bit` field, the struct uses `u8`, an unsigned 8-bit integer. The reason is that a corrupted `bool` can already lead to undefined behavior (e.g. when it has the value `3`), but an `u8` is always valid. This makes it possible to convert the `self` raw pointer into a reference at the beginning of the `is_consistent` implementation without invoking undefined behavior.

To check for bit-wise consistency, the implementation of `is_consistent` first extracts the parity bit from the last bit of the `parity_bit` field. It then counts

```
pub unsafe auto trait RestoreSafe {}
```

Listing 4.3: Definition of the `RestoreSafe` Trait

the 1-bits in the binary representation of `value`, adds the parity bit, and verifies that the sum is even. This makes it possible to detect all even numbers of bit-flips. There is no way to restore the original value with a single parity bit, therefore the `is_consistent_restore` function just calls into `is_consistent`.

The `ConsistencyCheckable` trait also works with other ECC variants, such as hamming codes or CRC checksums. For our framework, we use a sum-and-copy ECC variant, whose implementation is described in Section 4.5.

4.4 The `RestoreSafe` Trait

This section describes the implementation of the `RestoreSafe` trait presented in Section 3.3. The purpose of the trait is to mark types that are guaranteed to be pointer consistent after a reboot. Such types can either be types without any reference or pointer fields, or types for that it is guaranteed at compile time that each reference or pointer field points to the NVRAM. The `RestoreSafe` trait is implemented as an auto trait (see Section 2.1.2). As all auto traits, it is a marker trait without any methods.

Listing 4.3 shown the definition of the trait. It is declared as `unsafe` because manual implementations have the potential to break memory safety in combination with our framework. For example, a manual implementation for a type that contains a pointer to the normal heap can result in an use-after-free vulnerability if the type is placed in NVRAM and restored after a reboot. By declaring the trait as `unsafe`, the programmer has to specify the `unsafe` keyword for each manual positive implementation, indicating a potential dangerous operation.

4.4.1 Negative Implementations

Auto traits are implemented for all types by default. Since pointer and reference types are potentially dangerous because they can point to an arbitrary memory location, we specify the negative implementations shown in Listing 4.4. Shared (`&T`) and exclusive references (`&mut T`) are separate types in Rust, so we provide negative implementations for both. The implementations are generic over all possible lifetimes and pointed types so that they apply to all possible reference types. Likewise, we add negative implementations for the read-only and mutable raw pointer types `*const T` and `*mut T`. The `unsafe` keyword is not needed for these implementations since it is only required for positive implementations.

```
impl<'a, T> !RestoreSafe for &'a T {}
impl<'a, T> !RestoreSafe for &'a mut T {}
impl<T> !RestoreSafe for *const T {}
impl<T> !RestoreSafe for *mut T {}
```

Listing 4.4: Negative `RestoreSafe` Implementations for Pointer and Reference Types

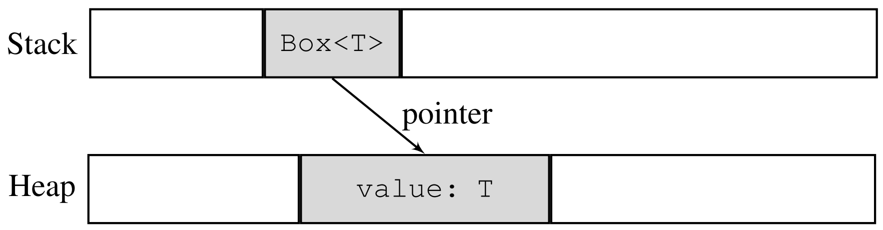

Figure 4.3: The Memory Layout of a Heap Allocated Value

The result of these negative implementations is that the trait is only implemented for types without any pointer or reference fields. This also rules out any collection types that can dynamically grow, such as vectors or linked lists. To make this limitation less severe, positive implementations are provided for selected collection types that are guaranteed to use the NVRAM as backing storage.

4.4.2 Positive Implementations

Ideally, we would like to provide positive implementations for all pointers and references that point to the NVRAM. Unfortunately, pointers and references contain no type information about the storage location, so that this approach is not possible. However, there is a class of types that contains this information, albeit often implicitly: types that perform heap allocations.

The simplest heap allocation type is `Box<T>`, which represents a heap allocated value of type `T`. When a `Box` is created, for example through the `Box::new` function, it allocates the required memory from the heap and internally keeps a pointer to the memory location. Figure 4.3 shows the memory layout for a `Box<T>` instance that lives on the stack. We see that the actual value is allocated on the heap. When the `Box` goes out of scope, its destructor frees the heap allocated value again.

The destructor can only safely free the heap-allocated value because it is guaranteed that the inner pointer of a `Box<T>` always points to the heap. This means that the storage location of the `Box<T>` pointer is known at compile time. However, we cannot provide a positive `RestoreSafe` implementation for `Box<T>`

```
unsafe impl<T, A> RestoreSafe for Box<T, A>
    where T: RestoreSafe, A: RestoreSafeAllocator
{}

pub unsafe trait RestoreSafeAllocator: Alloc {}
```

Listing 4.5: Positive `RestoreSafe` Implementation for `Box`

because it points to the volatile heap instead of the NVRAM. Fortunately, it is possible to let the `Box` type use the NVRAM instead by using a *custom allocator*.

Custom Allocators

Currently, the `Box` type is limited to the global heap allocator, but there is an accepted proposal [75] and and implementation pull request [76] to make `Box` and other heap allocation types work with custom allocator types. The idea is to add a second type parameter `A` to the `Box` type that represents the allocator that should be used for the allocation and deallocation. This makes it possible to create a `Box<T, A>` type that uses a custom allocator that allocates from the NVRAM instead of the normal heap. For usability and compatibility reasons, this type parameter defaults to the global allocator, so that `Box<T>` can be used just as before.

Since the used allocator is a generic parameter to the `Box` type, it is statically known at compile time. This allows us to provide a positive `RestoreSafe` implementation for `Box` types that use the NVRAM as backing storage. Listing 4.5 shows this positive implementation. It is generic over all possible allocators, but uses a `where` bound to enforce the following constraints:

- First, the stored value type must be `RestoreSafe` itself, similar to the default transitive behavior of auto traits. This is important because the value type could contain inner pointers itself.

- Second, the allocator parameter must implement a special marker trait called `RestoreSafeAllocator`, which abstracts over different NVRAM allocators. The trait extends the `Alloc` trait of the standard library, which is the trait that `Box` requires for the allocator parameter. Since the programmer must ensure to only implement the `RestoreSafeAllocator` trait for allocators that use the NVRAM as backing storage, the trait is `unsafe` to implement.

Implementing Custom Allocator Support in the Standard Library

While the proposal for custom allocator support for the heap allocation types of the standard library was already accepted in 2016 [75], it is still not implemented at the time of writing this thesis. Since this custom allocator support is required for the positive `RestoreSafe` implementations for `Box` and similar types, we created an modified version of the standard library. Building on top of the current work in progress pull requests for custom allocator support [76], we implemented custom allocator support for the `Box`, `Vec` (a heap allocated growable array), `VecDeque` (a heap allocated growable ring buffer), `Arc` (a heap allocated value that uses atomic reference counting), and `BTreeMap` (a tree-based, heap allocated key-value map) types.

To use the modified version of the standard library, we utilized that Redox uses the `xargo` [77] tool to rebuild the standard library with a modified feature set (e.g. disabling backtraces on unsupported platforms). This allowed us to build a custom version of the standard library by making `xargo` use our modified version of `libstd`.

The dependence on a custom standard library version is a strong drawback of our solution. However, given that there is already an accepted proposal and a draft pull request for custom allocator support that are very similar to our custom implementation, we believe that the migration of our framework to the future official implementation will be manageable.

Positive `RestoreSafe` implementations for other Allocation Types

In addition to `Box`, our framework provides positive implementations for the other allocation types with custom allocator support. The implementations are shown in Listing 4.6. Like the implementation for `Box`, the implementations require that the value type is `RestoreSafe` and that the allocator parameter implements the `RestoreSafeAllocator` trait.

For `BTreeMap`, the allocator type must additionally implement the `Default` trait and its associated error type `Err` must implement the `Debug` trait. Both traits are required only because of our implementation choices when implementing custom allocator support in the standard library. Namely, the `Default` trait is used as a way of creating an allocator instance without needing access to an existing allocator instance. This made the implementation much easier since it is not necessary to store an additional allocator reference in every tree node this way. The `Debug` bound on the error type is required because we decided to panic on allocation failures, also in order to simplify the implementation. It is worth to note that the future official implementation likely will not have this limitations.

```
unsafe impl<T, A> RestoreSafe for Vec<T, A>
    where T: RestoreSafe, A: RestoreSafeAllocator
{}

unsafe impl<T, A> RestoreSafe for Arc<T, A>
    where T: RestoreSafe, A: RestoreSafeAllocator
{}

unsafe impl<K, V, A> RestoreSafe for BTreeMap<K, V, A>
    where K: RestoreSafe,
          V: RestoreSafe,
          A: RestoreSafeAllocator + Default,
          A::Err: Debug
{}
```

Listing 4.6: Positive `RestoreSafe` Implementations for Supported Allocation Types

```
pub struct ContextList {
    map: BTreeMap<ContextId, Arc<RwLock<Context>>>,
    next_id: usize
}
```

Listing 4.7: Definition of the `ContextList` Type

4.4.3 Example: Applying `RestoreSafe` to Redox

The `RestoreSafe` trait ensures that no violations of pointer consistency can occur when partially restoring state after a system reboot. By providing positive implementations for collection types that are guaranteed to be stored in NVRAM, our framework is applicable to many real-world use cases.

As an example, Listing 4.7 shows the definition of the `ContextList` struct of the Redox kernel. This type is responsible for keeping track of all processes and threads, similar to process and thread control blocks in Linux. The `map` field is a `BTreeMap` that maps a `ContextId` to an atomically reference counted `Context` instance protected by a read-write lock. For this example, we treat `ContextId` as a normal integer and ignore the definition of the `Context` type.

Since the `BTreeMap` and the `Arc` wrapper allocate memory from the volatile heap by default, the `ContextList` type is not `RestoreSafe`. The exact reason that the trait is not implemented is that both collection types contain internal pointers to which the negative implementations apply. Further, no positive implementation applies because the second type parameter defaults to the system allocator, which allocates from the volatile heap and thus does not implement the `RestoreSafeAllocator` trait.

```
pub struct ContextList {
    map: BTreeMap<ContextId,
        Arc<RwLock<Context>, PersistentRamAllocator>,
        PersistentRamAllocator>,
    next_id: usize
}
```

Listing 4.8: Definition of the `ContextList` Type, Adjusted to Implement the `RestoreSafe` Trait

To implement the `RestoreSafe` trait for `ContextList`, we need to place the internal allocations in NVRAM instead. This can be done by supplying a `PersistentRamAllocator` as a second generic parameter to both the `BTreeMap` and the `Arc` types. This allocator is provided by our NVRAM framework. It is similar to the default allocator, but it allocates memory from the NVRAM instead of the volatile heap.

Listing 4.8 shows the adjusted definition of the `ContextList` type. Now both the `BTreeMap` and the `Arc` types allocate their memory from NVRAM so that their content stays valid across reboots. Since the `PersistentRamAllocator` implements the `RestoreSafeAllocator` trait, the positive implementations of `RestoreSafe` apply. As a result, the `ContextList` is now `RestoreSafe` too (under the assumption that `Context` is `RestoreSafe`).

4.4.4 Stability

Apart from a custom version of the standard library, the `RestoreSafe` trait relies on the *auto traits* language feature. This feature is still unstable and might change in the future. For this reason it is only available when using a nightly version of the Rust compiler together with the `#![feature(optin_builtin_traits)]` feature gate.

While this usage of an unstable feature limits the stability of our framework, we believe that the *auto traits* will continue to exist and not receive any major changes in the future because it is the base for the `Send` and `Sync` traits, which are the fundamental building block of Rust's compile time thread thread safety. Further, Redox OS already relies on a nightly compiler and unstable features even without our framework, so that the stability of our framework is not the critical factor.

4.5 Error Correcting Codes

As described in Section 3.2, our framework employs error correcting codes (ECCs) to ensure bit-wise consistency of (restored) values. Instead of using compute intensive ECCs such as Hamming or Reed-Solomon codes, we chose a sum-and-copy ECC for our framework based on an evaluation from related work [64].

To give the programmer full control of ECCs recalculations and allow the gradual transformation of systems, we use an explicit approach for adding ECCs to values, as described in Section 3.2.2. In this section, we describe the implementation of wrapper types that add an ECC to values in an explicit way.

This section is structured into subsections as follows. Section 4.5.1 describes the implementation of a general `Ecc` wrapper type that augments an arbitrary type with an ECC. In Section 4.5.2 we then describe the implementation of the `ConsistencyCheckable` trait for the `Ecc` wrapper type. To also support consistency checks for types with internal pointers, Section 4.5.3 explains how our framework supports nested consistency checks. Finally, Section 4.5.4 describes how required code modifications can be minimized by integrating ECCs into common wrapper types such as `RwLock`.

4.5.1 The `Ecc` Wrapper Type

Instead of individually adding error correcting codes to all types that might be stored in NVRAM, we implement an `Ecc` wrapper type that protects an arbitrary type with an error correcting code. This has the advantage that the types themselves require no modification. It also allows to add the checksum only where it is necessary, thereby limiting overhead.

The definition of the `Ecc` type is shown in Listing 4.9. The structure is generic over a value type `T` to allow wrapping an arbitrary type. Since it is unknown whether the `value` and `copy` instances are bit-wise consistent before checking, the fields do not store a `T` instance directly. Instead, they use the `MaybeUninit` wrapper type of the standard library, which is an abstraction for possibly uninitialized memory that prevents the compiler from making any assumptions about the wrapped type (see Section 2.1.4). This ensures that no undefined behavior is invoked when a bit-flip occurs.

Figure 4.4 shows the memory layout of the `Ecc<T>` type using the `field_name: FieldType` syntax. We see that the value more than doubles in size. On the other hand, it can be computed very efficiently by the CPU, so it has a lower performance impact than other ECC variants with a lower memory overhead such as Hamming codes.

```
struct Ecc<T> {
    value: MaybeUninit<T>,
    sum: u32,
    copy: MaybeUninit<T>,
}
```

Listing 4.9: Definition of the `Ecc` Wrapper Type

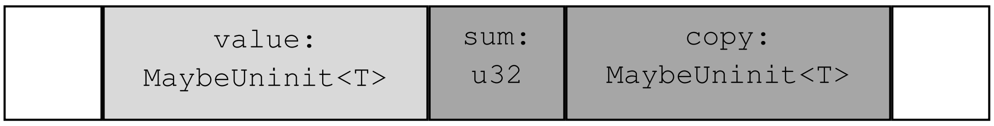

Figure 4.4: Memory Layout of the `Ecc<T>` Type

Apart from a `new` constructor function that wraps a given value, the `Ecc` type provides a `get_ref` method that returns a reference to the wrapped value after checking its consistency. If the consistency check fails, the function panics. There is also a `try_get_ref` method, that returns a `Result` instead of panicking, and an unsafe `get_ref_unchecked` method that allows to completely skip the consistency check for improving performance, which should only used when an inconsistent value is extremely unlikely.

Mutating a protected value is more complicated because the checksum needs to be updated for each modification. The `Ecc` type provides a `get_mut` method for modifying the wrapped value. Instead of returning a `&mut` reference to the value, the method returns an `EccGuard` struct that recalculates the checksum when its destructor is called. By implementing the `Deref` and `DerefMut` traits, the struct can be used just like a normal `&mut` reference in most cases.

Listing 4.10 shows an usage example for the `Ecc` wrapper type. In line 2, a new `Ecc` instance is created, wrapping the value `0` of type `u32`. In line 3, the current value of the `number` variable is printed by using the `get_ref` method, which returns a shared reference to the wrapped value after checking for consistency. Line 4 uses the `get_mut` method for retrieving an `EccGuard` instance that allows mutation of the wrapped value. Here, we increase the value by one. Since it is not assigned to a variable, the `EccGuard` instance only lives for the single statement and immediately goes out of scope after the value was increased. At this point, the destructor of the `EccGuard` type runs, which recalculates the `sum` and `copy` fields.

In line 6, a new `EccGuard` instance is created by calling `get_mut` again. This time the result is assigned to the variable `i`, so it lives until the last usage of `i` in line 10. Using this pattern, the wrapped value can be repeatedly modified without recalculating the checksum every time. This way, performance can be

```
1  fn test() {
2      let mut number = Ecc::new(0u32);
3      println!("Current value: {}", number.get_ref());
4      number.get_mut() += 1;
5
6      let i = number.get_mut();
7      if *i > 1 {
8          *i = 10;
9      }
10     *i += 1;
11 }
```

Listing 4.10: Example Usage of the `Ecc` Wrapper Type

improved at the cost of slightly decreasing reliability. For example, a bit-flip in line 8 would not be detected. It is up to the programmer to find a good tradeoff between reliability and performance.

4.5.2 Implementing the `ConsistencyCheckable` Trait

To make the `Ecc` type usable together with our NVRAM framework, it needs to implement the `ConsistencyCheckable` trait. However, the trait can only be safely implemented when the wrapped type contains no reference or pointer fields. The reason is that the checksum only protects the pointer itself, but not the pointed value. The `ConsistencyCheckable` trait, however, requires that the whole value is checked for bit-wise consistency, including values behind pointers.

For this reason, the trait is only implemented for types that implement a `NoInternalPointers` marker trait. This trait is very similar to the `RestoreSafe` trait, but without any of the positive implementations. Thus, the trait is implemented for exactly the types without pointer fields. By requiring the `NoInternalPointers` trait for the wrapped type, only `Ecc` instances without internal pointers can implement the `ConsistencyCheckable` trait.

Listing 4.11 shows the trait implementation in Rust-like pseudo-code. The `is_consistent` method is not allowed to modify the value, so it only checks if the stored 32-bit sum is still valid for the value without attempting any restoration. The `is_consistent_restore` method additionally checks if the sum is valid for the copy and if `value` and `copy` are equal. In the former case, it assumes that the copy is still consistent and restores the value from it. In the latter case, the method assumes that the sum itself was corrupted, so it recalculates it. If no check succeeds, the value is not restorable and false is returned.

```
unsafe impl<T> ConsistencyCheckable for Ecc<T>
where
    T: NoInternalPointers,
{
    fn is_consistent(self: *const Self) -> bool {
        self.sum == calculate_sum(self.value)
    }

    fn is_consistent_restore(self: *mut Self) -> bool {
        if self.sum == calculate_sum(self.value) {
            return true;
        }
        if self.sum == calculate_sum(self.copy) {
            self.value = self.copy;
            return true;
        }
        if self.value == self.copy {
            self.sum = calculate_sum(self.value);
            return true;
        }
        false
    }
}
```

Listing 4.11: Implementation of the `ConsistencyCheckable` Trait for the `Ecc` Wrapper Type (in Pseudo Code)

```
pub unsafe trait NestedConsistencyChecks {
    fn check_consistency_nested(self: *const Self) -> bool;

    fn check_consistency_nested_restore(self: *mut Self)
        -> bool;
}
```

Listing 4.12: Definition of the `NestedConsistencyChecks` Trait

4.5.3 Nested Consistency Checks

As explained in the previous section, the `Ecc` wrapper type does not support types with internal pointers. The reason is that the `ConsistencyCheckable` trait requires that the whole type is checked for bit-wise consistency, including values behind pointers, but the `Ecc` type only checks the `Self` instance. This makes the `Ecc` type unsuitable in many cases. For example, types with internal pointers are common in the Redox kernel, for example in form of a `BTreeMap` inside the `SchemeList` type, which keeps track of all available schemes.

To fix this issue, our framework provides an additional `NestedEcc` type for types with pointer fields. In combination with a `NestedConsistencyChecks` trait, this type recursively checks the consistency of all values behind pointers, in order to ensure the bit-wise consistency of the complete value.

The `NestedConsistencyChecks` Trait

The `NestedConsistencyChecks` trait describes types that contain one or more internal pointers to checksum-protected values, such as `Box<Ecc<T>>`. Checking the bit-wise consistency of the outer type does not suffice for these types. Instead, the consistency of the value behind the pointer must be checked too. The `NestedConsistencyChecks` trait makes it possible to do that.

The definition of the trait is shown in Listing 4.12. Is is almost identical to the `ConsistencyCheckable` trait (shown in Listing 4.1). The differences between the two traits are the method names and that they are called in different contexts: The methods of the `NestedConsistencyChecks` trait are called with `Self` value that was already checked for bit-wise consistency, while the methods of the `ConsistencyCheckable` trait are called with a potentially corrupted `Self` value.

The purpose of the methods of the `NestedConsistencyChecks` trait is to check the bit-wise consistency of values behind pointers. As an example, Listing 4.13 shows the implementation of the trait for `Box<Ecc<T>>`. First, the methods convert the `self` raw pointer into a reference type. This is safe in this case because the bit-wise consistency of the `Box` wrapper was already checked and the

```
unsafe impl<T> NestedConsistencyChecks for Box<Ecc<T>>
    where T: NoInternalPointers
{
    fn check_consistency_nested(self: *const Self) -> bool {
        let s = unsafe { &*self };
        ConsistencyCheckable::is_consistent(s.deref())
    }

    fn check_consistency_nested_restore(self: *mut Self)
        -> bool
    {
        let s = unsafe { &mut *self };
        ConsistencyCheckable::is_consistent_restore(
            s.deref_mut())
    }
}
```

Listing 4.13: Implementation of `NestedConsistencyChecks` for `Box`

`Ecc` type behind the pointer uses `MaybeUninit`, so that the compiler cannot make any assumptions about its value. In the second step, the methods call the corresponding methods of the `ConsistencyCheckable` trait on the `Ecc` instances behind the pointer.

The `NestedEcc` Type

Wrapping the `Box<Ecc<T>>` type into the `Ecc` wrapper type does not work because `Box` has an internal pointer and thus does not implement the required `NoInternalPointers` trait. As an alternative, our framework provides a `NestedEcc` wrapper type that is identical to the `Ecc` type, but uses the `NestedConsistencyChecks` trait instead of the `NoInternalPointers` trait as a bound for the `ConsistencyCheckable` implementation.

Listing 4.14 shows the implementation of the `ConsistencyCheckable` trait for the `NestedEcc` type. The `is_consistent` method first checks the consistency of the `Self` type by comparing the sum, similar to the implementation for `Ecc` (see Listing 4.11). If the `Self` type is consistent, the `check_consistency_nested` method of the `NestedConsistencyChecks` trait is invoked with the wrapped value as argument. The `check_consistency_nested` method then recursively performs ECC checks for values behind inner pointers. The implementation of the `is_consistent_restore` method is not shown here for space reasons. It is very similar, with the difference that it attempts to restore a corrupted `Self` value and calls into `check_consistency_nested_restore`.

By combining the `Ecc` and `NestedEcc` wrapper types, it is possible to re-

```
unsafe impl<T> ConsistencyCheckable for NestedEcc<T>
where
    T: NestedConsistencyChecks,
{
    fn is_consistent(self: *const Self) -> bool {
        if self.sum == calculate_sum(self.value) {
            NestedConsistencyChecks::check_consistency_nested(
                self.value.as_ptr())
        } else { false }
    }

    fn is_consistent_restore(self: *mut Self) -> bool { ... }
}
```

Listing 4.14: `ConsistencyCheckable` Implementation for `NestedEcc`

Step	Operation	on Type	Code in
1	`is_consistent()`	outer `NestedEcc`	Listing 4.14
2	`check_consistency_nested()`	outer `Box`	Listing 4.13
3	`is_consistent()`	inner `NestedEcc`	Listing 4.14
4	`check_consistency_nested()`	inner `Box`	Listing 4.13
5	`is_consistent()`	`Ecc<T>`	Listing 4.11

Table 4.1: Steps for Checking Consistency of the Nested Type `NestedEcc<Box<NestedEcc<Box<Ecc<T>>>>>`

cursively check the bit-wise consistency of complex nested types. As an example, Table 4.1 shows the sequence of consistency checks that occur for the `NestedEcc<Box<NestedEcc<Box<Ecc<T>>>>>` type. For each step, the table shows the method that is being invoked, the type on which it is invoked, and a reference to the listing that shows the method implementation.

In step 1, an outside caller such as the persistent RAM manager calls the `is_consistent` method of the `ConsistencyCheckable` trait. The implementation for the outer `NestedEcc` type then verifies the consistency of the outer `Box` instance. If it is consistent, it calls in step 2 the `check_consistency_nested` method of the `NestedConsistencyChecks` trait on the wrapped `Box` instance. This method then calls the `is_consistent` method on the value behind the pointer (step 3). The wrapped value is a `NestedEcc` instance again, so it again calls `check_consistency_nested` on the wrapped `Box` instance after checking the consistency of `Self` (step 4). Finally, the inner `Box` instance calls `is_consistent` on the wrapped `Ecc` value, which has no further internal pointers.

```
pub struct BufferField {
    pub source_buf: Box<AmlValue>,
    pub index: Box<AmlValue>,
    pub length: Box<AmlValue>
}
```

Listing 4.15: The `BufferField` Type of the Redox Kernel

```
pub struct EccBufferField {
    pub source_buf: Box<Ecc<AmlValue>>,
    pub index: Box<Ecc<AmlValue>>,
    pub length: Box<Ecc<AmlValue>>
}
```

Listing 4.16: ECC-Augmented Version of the `BufferField` Type

Deriving `NestedConsistencyChecks`

While our framework provides `NestedConsistencyChecks` implementations for supported collection types, these collection types are often not used directly. Instead, they are commonly wrapped in structs and other compound types. For example, the `ContextList` struct of the Redox kernel (shown in Listing 4.7) internally uses a `BTreeMap` collection type for its `map` field. Since our framework can only provide `NestedConsistencyChecks` implementations for known types, the programmer needs to implement the trait themselves in order to use custom types such as `ContextList` together with the `NestedEcc` type.

As an example, Listing 4.15 shows the definition of the `BufferField` struct of the Redox kernel, which is used as part of the parser for the ACPI machine language (AML). To allow checking the consistency of the struct, it should be wrapped in a `NestedEcc` type. For that, the values stored in the `Box` types need to be protected by introducing `Ecc` wrapper types. The ECC-augmented version of `BufferField` is shown in Listing 4.16.

In the second step, the `NestedConsistencyChecks` trait must be implemented for `EccBufferField` because it is required by the `NestedEcc` type. Our framework cannot provide this implementation because it does not know the `EccBufferField` type. Thus, the programmer needs to manually provide an implementation.

Listing 4.17 shows this manual implementation. First, both methods convert the raw pointer into a reference. This operation is safe here because of two reasons. First, the `Self` type is checked for consistency before the methods are called. Second, all the pointer fields wrap the pointed value in an `Ecc` type, which uses `MaybeUninit` and thus prevents the compiler from making assumptions

```
unsafe impl NestedConsistencyChecks for EccBufferField {
    fn check_consistency_nested(self: *const Self) -> bool {
        let s = unsafe { &*self };
        s.source_buf.check_consistency_nested() &&
        s.index.check_consistency_nested() &&
        s.length.check_consistency_nested()
    }

    fn check_consistency_nested_restore(self: *mut Self)
        -> bool
    {
        let s = unsafe { &mut *self };
        s.source_buf.check_consistency_nested_restore() &&
        s.index.check_consistency_nested_restore() &&
        s.length.check_consistency_nested_restore()
    }
}
```

Listing 4.17: Manual Implementation of `NestedConsistencyChecks` for `EccBufferField`

about the wrapped value. After converting the raw pointer to a reference, the methods recursively invoke themselves on each pointer field. The methods only return `true` if all fields are consistent.

While such a manual implementation of the `NestedConsistencyChecks` trait is possible, it has several drawbacks. First, it is cumbersome for the programmer because such an implementation is required for every struct that should be wrapped in a `NestedEcc` type. Second, it is prone to programmer mistakes. For example, copy-and-paste errors and misspellings are common problems in such boilerplate code. More importantly, code refactorings that change the struct type could lead to an outdated implementation. For example, the programmer can easily forget to update the implementation in Listing 4.17 after adding a fourth `Box` field to the `EccBufferField` type. As a result, the newly added field would not be checked for consistency, which can result in undefined behavior when the field is accessed after a restart.

For these reasons, our framework provides a way to automatically implement the `NestedConsistencyChecks` type for custom structs. The idea is to provide a procedural *derive macro* (see Section 2.1.3) that auto-generates the implementation. Listing 4.18 shows how the macro can be used to replace the manual `NestedConsistencyChecks` with a simple `#[derive]` attribute.

The derive macro works by reading the source code of the annotated struct. It then generates the source code for the trait implementation by recursively calling the respective `NestedConsistencyChecks` method for each pointer field of the

```
#[derive(NestedConsistencyChecks)]
pub struct EccBufferField {
    ...
}
```

Listing 4.18: Automatic Implementation of `NestedConsistencyChecks`

struct. As a result, an automatic implementation equivalent to the implementation shown in Listing 4.17 is generated. Thus, the mentioned programmer mistakes are completely prevented.

4.5.4 Reducing Required Code Modifications

As explained in Section 4.5.1, the `Ecc` wrapper type requires explicit calls to its `get_ref` and `get_mut` methods in order to access the wrapped value. Thus, extensive code modifications are required when wrapping an existing type into an `Ecc` wrapper. For example, replacing the `BufferField` type shown in Listing 4.15 with the `EccBufferField` type shown in Listing 4.16 would require adding a `get_ref` call for every read operation on a field of the struct.

To reduce the required code modifications, our framework provides ECC-augmented versions of common wrapper types such as `RwLock`. By keeping the exact same API as the augmented types, the required code modifications are minimized. In the following, we focus on the `RwLock` type for simplicity, but the same approach can be also applied to other wrapper types.

Listing 4.19 shows an example code that uses the `RwLock` synchronization primitive, which provides a read-write-lock. The type allows an arbitrary number of readers or a single writer at any point in time, but not both. Through the `read` method, the `RwLock` is locked in shared access mode. In this mode, other `read` calls succeed, but `write` calls are blocked until the last read operation finishes. The `read` method returns a shared reference to the wrapped value, which only allows reading the value but prevents any modification. In contrast, the `write` method locks the `RwLock` with exclusive write access, which blocks all other `read` and `write` calls as long the lock is held. This way, only a single writer can be active at any point in time, which makes it safe to modify the wrapped value through a `&mut` reference.

The `RwLock` type is common in multi-threaded programs. For example, the Redox kernel uses `RwLock` types for managing access to the global `Context` types stored in the `ContextList` (see Listing 4.7). By creating an `EccRwLock` wrapper type that augments the `RwLock` type with an ECC, we can protect the items of the `ContextList` without requiring any modifications to code that accesses the list.

```
let lock = RwLock::new(0);

let value_0 = lock.read();
let value_1 = lock.read();
assert_eq!(value_0, value_1);

let mut value_mutable = lock.write();
*value_mutable = 10;
```

Listing 4.19: Example Use of `RwLock`

```
pub struct EccRwLock<T> {
    inner: RwLock<Ecc<T>>
}

impl<T> EccRwLock<T> {
    pub fn read(&self) -> EccRwLockReadGuard<T> {
        let ecc = self.inner.read();
        assert!(ecc.is_consistent());
        EccRwLockReadGuard(ecc)
    }

    pub fn write(&self) -> EccRwLockWriteGuard<T> {
        let mut ecc = self.inner.write();
        assert!(ecc.is_consistent_restore());
        EccRwLockWriteGuard(ecc)
    }
}
```

Listing 4.20: Definition of `EccRwLock`

Listing 4.20 shows the definition of the `EccRwLock` type, which is just a wrapper around `RwLock<Ecc<T>>`. Like `RwLock`, the type supports `read` and `write` operations, but the operations additionally perform consistency checks. Restoration of inconsistent values is only possible for the `write` method because it requires a mutable reference. Instead of returning bare reference types, both methods return a *guard* type. This guard type can used like normal references, but has a destructor that recalculates the ECC checksum (only for `write`) and releases the lock when it is dropped.

In order to add an ECC protection to the code example in Listing 4.19, only the type name in the first line needs to be changed from `RwLock` to `EccRwLock`. The remaining code can remain unchanged because the `EccRwLock` type provides the same API.

```
#[restorable(restore = true)]
static S: StaticType = static_init_expression;
```

Listing 4.21: Original Static Definition

```
static S: _S_Wrapper = _S_Wrapper {
    value: spin::Once::INIT,
}

struct _S_Wrapper {
    value: spin::Once<&'static mut StaticType>,
}
```

Listing 4.22: Generated Static Definition

4.6 The `#[restorable]` Attribute

By default, static variables are stored in the `.data` or `.bss` sections of the executable, which is newly initialized when the executable is loaded again after a restart. To preserve the value of a static variable across reboots, it must be stored in the NVRAM instead, which is not reinitialized after a reboot.

One way to solve this problem is to modify the bootloader to directly load relevant sections into the NVRAM and omit reinitialization of the static value on a reboot. However, this would make the bootloader much more complex and the kernel dependent on a specific non-standard bootloader implementation.

As described in Section 3.5, we instead provide a `#[restorable]` attribute that lazily initializes the static variable in NVRAM on the first use. Listing 4.21 shows how the attribute can be applied to a static variable. The `static_init_expression` can be an expression of any form, ranging from a simple integer literal to a complex initialization block inside `{}` braces. Using the attribute, no bootloader modification is required so that the kernels stays portable. In the following, we describe the functionality of the `#[restorable]` attribute with an example.

The `#[restorable]` attribute is implemented as an attribute-like procedural macro, as described in Section 2.1.3. It transforms the `static` declaration shown in Listing 4.21 to the code shown in Listing 4.22. The `static` still has the same name `S`, but its type was changed to a newly defined `_S_Wrapper` structure, which has a single `value` field. Instead of storing the value of type `StaticType` directly, the field contains a reference to it, wrapped by the `Once` synchronization primitive. This makes it possible to allocate the value in the NVRAM and only keeping a pointer to it.

```
impl ::core::ops::Deref for _S_Wrapper {
    type Target = StaticType;

    fn deref(&self) -> &Self::Target {
        let allocate_static = || [...];
        self.value.call_once(allocate_static)
    }
}
```

Listing 4.23: Generated Deref Implementation

```
let allocate_static = || {
    let init = || static_init_expression;
    PERSISTENT_RAM_MANAGER.allocate_named(self as usize, init)
}
```

Listing 4.24: Generated `allocate_static` Implementation for `restore = true`

To make the generated static variable usable like a static variable of type `StaticType`, we provide an implementation of the `Deref` trait, which implements the dereferencing operation. Since Rust performs auto-dereferencing on method calls and field accesses, the `Deref` implementation makes the generated `S` static usable like a static of type `StaticType`.

Listing 4.23 shows the implementation of the `Deref` trait. The type alias `Target` specifies that the `_S_Wrapper` dereferences to a `StaticType` type. The `deref` method performs the actual dereferencing and is automatically called by the compiler. The method is implemented using the `call_once` method of the `Once` synchronization primitive, which allows running an one-time global initialization routine on the first `call_once` call. On subsequent calls, a reference to the already initialized value is returned.

The implementation of the `allocate_static` closure depends on the value of the `restore` argument passed to the `#[restorable]` attribute. The implementation for `restore = true`, is shown in Listing 4.24. In this case, the generated closure calls the `allocate_named` method of the global `PERSISTENT_RAM_MANAGER` instance, which is described in Section 3.1.1. The address of the static variable is hereby used as the unique key. A closure returning the `static_init_expression` is passed as an argument.

Listing 4.25 shows the implementation of the closure for `restore = false`. The expression is allocated on the normal heap using the `Box` type. This way, the `static` is newly initialized on every reboot, so that the behavior is similar to a `static` without the `restorable` attribute. The difference is that more com-

```
let allocate_static = || {
    Box::leak(Box::new(static_init_expression))
}
```

Listing 4.25: Generated `allocate_static` Implementation for `restore = false`

plex initialization expressions are possible with the attribute since the initialization happens at runtime instead of compile time.

Chapter 5

Evaluation

To evaluate our NVRAM framework, we applied it to both the kernel and user-space programs of Redox OS (see Section 2.2). As we did not have access to real NVRAM, we simulated it using QEMU. By injecting bit flips and forcing system reboots through QEMU's QMP interface, we evaluated the robustness of our solution.

We performed two types of experiments. First, we added an ECC protection to a kernel-internal data structure and evaluated the robustness by injecting random bit-flips. By measuring the progress of a test program we show that our ECC wrapper types considerably improve the resilience against flipped bits. Second, we implemented support for persistent file descriptors and evaluated them by causing system restarts while running a test program in userspace. By examining the debug output of the system we show that our framework is able to keep state across system restarts.

This chapter is structured as follows: First, we introduce the QEMU machine emulator in Section 5.1. We describe its QMP interface, which allows to control a guest instance from the host system, and show how it can be used to inject system restarts. We also present a QMP-based fault injection framework for inducing bit-flips and explain how NVRAM can be simulated in QEMU.

In Section 5.2, we then evaluate the usability and robustness of the ECC wrapper types of our framework by protecting a kernel-internal data structure of the Redox kernel. We show that the use of the combined wrapper types described in Section 4.5.4 reduces the required code modifications to a minimum. By inducing random bit-flips through the QMP protocol, we show that the ECC protection of our wrapper types considerably increases the robustness of the protected data structure. We also evaluate the performance overhead of the ECC wrapper types by running microbenchmarks.

Finally, Section 5.3 shows how we applied our NVRAM framework to implement support for persistent file descriptors in Redox. By injecting system reboots

```
{ "execute": "system_reset" }
```

Listing 5.1: QMP-Command for Performing a Hard Reset of the Guest System

while running an userspace test program, we show that persistent file descriptors allow the test program to continue using an opened file across system restarts. We also perform an analysis of error cases for different restart intervals to show the robustness of our implementation.

5.1 QEMU

QEMU [78] is an open source machine emulator and virtualizer. It is implemented as a hosted virtual machine monitor (VMM) that uses binary translation to run guest systems in a virtual machine. Apart from `x86`, it supports many other system architectures such as `ARM` or `RISC-V` [79]. By utilizing the *kernel-based virtual machine* (KVM) of the Linux kernel, near native performance is possible [80].

Apart from a high performance, QEMU provides useful debugging capabilities. For example, it supports logging of system interrupts and can connect to an external instance of the *GDB* or *LLDB* debuggers. Through the built-in *monitor*, full introspection and control of the system state is possible [81].

For our evaluation, we use the *QEMU machine protocol*, which allows to access and modify system state from scripts on the host system. By injecting system restarts and bit-flips into the guest system, we simulate the characteristic properties of NVRAM without having access to a real NVRAM device.

5.1.1 The QEMU Machine Protocol

The *QEMU machine protocol* (QMP) is a JSON-based interface that allows to control a QEMU instance from the host system [82]. The protocol supports a wide range of commands, for example for dumping the guest memory or for simulating a key-press event of the keyboard device.

To evaluate our NVRAM framework, we require two kinds of functionality. First, we need a way to force a system reset in order to simulate a power outage or device failure. This is possible using the `system_reset` command, which is shown in Listing 5.1. Second, we need to be able to corrupt main memory, to simulate flipped bits that can occur in NVRAM. Since this functionality is not supported in the upstream version of QEMU, we used a QEMU fork from Xilinx [83] instead, which provides a fault injection framework.

```python
import fault_injection
import sys
import random

fi = None # fault injector

def main():
    global fi
    fi = fault_injection.FaultInjectionFramework(sys.argv[1], 0)

    fi.notify(1 * 1000 * 1000 * 1000, corrupt) # after 1 second

    fi.run()
    sys.exit(1)

def corrupt():
    byte_number = random.randint(0, 20 * 1024 * 1024)
    byte = 0x11
    fi.write(byte_number, byte, 1, 0)

    fi.notify(1 * 1000 * 1000, corrupt) # every millisecond

if __name__ == '__main__':
    main()
```

Listing 5.2: Python Script that Corrupts a Random Memory Byte Every Millisecond

5.1.2 Fault Injection

The fault injection framework of the QEMU fork by Xilinx modifies the QMP interface to allow arbitrary memory modifications through Python scripts. For example, Listing 5.2 shows a script to set a random memory byte to `0x11` every millisecond. First, the fault injection framework is initialized. Then, the `notify` method is used to register the `corrupt` function as a callback after one second. By invoking the `run` method, the fault injection loop is started. When called, the `corrupt` function chooses a random byte in the first 20 megabytes of memory and overwrites it with `0x11` through the `write` method. Afterwards, it uses the `notify` method to run itself again after one millisecond.

It is worth noting that the `write` method operates on physical addresses. To also support the targeted corruption of certain virtual memory areas, we implemented a `virt_to_phys` function that translates a virtual to physical address by traversing the guest's page tables.

```
pub struct ContextList {
    map: BTreeMap<ContextId, Arc<RwLock<Context>>>,
    next_id: usize
}
```

Listing 5.3: Definition of the `ContextList` Type

5.1.3 Simulating NVRAM

Using the fault injection framework and the QEMU machine protocol, we were able to simulate unreliable memory and cause system restarts at a selected point in time. In order to simulate the persistence of NVRAM, we utilized the fact that QEMU leaves the memory content unchanged when a reboot is injected via QMP. This makes any physical memory region a possible NVRAM region.

5.2 ECC Evaluation

To evaluate the robustness and the usability of our `Ecc` wrapper type, we used it to protect the kernel-internal `ContextList` data structure, which is shown in Listing 5.3. The data structure keeps track of the threads of all processes, comparable to the thread and process control blocks in Linux.

Internally, the `ContextList` struct uses a `BTreeMap` collection to keep track of all `Context` instances in the system. The `Context` type is a large struct that contains several types of IDs (process ID, user ID, group ID, etc.), a list of open file descriptors, a reference to the thread-local storage, the backed-up CPU state for paused threads, and many more fields.

The `ContextList` type is a good candidate for evaluating our ECC wrapper types in kernel context because it keeps a large portion of kernel-internal state and is regularly accessed. For example, on each context switch, several fields of the current `Context` instance are modified, such as the field that contains the current CPU state.

Our evaluation consists of the following parts: First, we study the number of required code changes for introducing an explicit ECC protection and compare the results for the `Ecc` and `EccRwLock` types. Second, we evaluate the robustness of the `EccRwLock` type by inducing random bit-flips. Finally, we perform a microbenchmark to evaluate the performance overhead of the `Ecc` wrapper type.

5.2.1 Required Kernel Changes

As we noted in Section 3.2.2, our framework uses an explicit approach with a non-transparent ECC. Compared to transparent ECCs, it has the advantage that all

Modified Lines	`RwLock<Ecc<Context>>`	`EccRwLock<Context>`
type renames	15	15
imports	5	5
value creation	1	0
value usage	100	0
total	123	20

Table 5.1: Number of Modified Lines

performance overhead is explicit to the programmer, which allows potential optimizations such as batching checksum recalculations. However, it requires code modifications whenever a new type should be protected by an ECC. To reduce these code modifications as much as possible, our framework provides special types that augment common wrapper type such as `RwLock` with an ECC, as explained in Section 4.5.4. The types have the same API as the normal, unprotected wrapper type, so that the needed code modifications are considerably reduced.

To evaluate how well this approach helps to reduce the number of required code changes, we used the `EccRwLock` type presented in Section 4.5.4 to protect the items of the central `ContextList` data structure of the Redox kernel. For that, we changed the item type of the `ContextList` from `Arc<RwLock<Context>>` to `Arc<EccRwLock<Context>>` and recorded the number of required code changes in the rest of the code.

For comparison, we also recorded the number of required changes for our general purpose `Ecc` wrapper type presented in Section 4.5.1. For that we repeated the same process, but switched the item type from `Arc<RwLock<Context>>` to `Arc<RwLock<Ecc<Context>>>`, which effectively adds the same kind of ECC-protection.

Table 5.1 compares the resulting number of code changes, both in total and categorized into different types of code changes. The left column shows the required number of code modifications for adding an additional `Ecc` wrapper to the type, so that the item type becomes `Arc<RwLock<Ecc<Context>>>`. The right column shows the number of required modifications when the `RwLock` is replaced by an `EccRwLock`, forming the type `Arc<EccRwLock<Context>>`.

The table shows that the combined `EccRwLock` type requires considerably less modifications than the `Ecc` type. When comparing the types of modifications, the table shows that the number of required type renames are the same, which makes sense since all mentions of the old type name must be replaced by the new type name, independently of the name of the new type. Also, both approaches require the same number of new imports, for either the `Ecc` or the `EccRwLock` type. For the `RwLock<Ecc<Context>>`, the expression for creating a new instance needs to change from `RwLock::new(...)` to `RwLock::new(Ecc::new(...))` in the single place where a new value is constructed. For the combined `EccRwLock` type the initialization syntax remains the same, apart from renaming the type.

The most notable difference occurs for the lines where the value is used. Introducing an additional `Ecc` wrapper requires additional `get_ref` or `get_mut` calls after locking, which requires the modification of 100 lines in our case. For the `EccRwLock` type, these modifications are not necessary since the type introduces the ECC transparently without changing the public API. It still calls the `get_ref` and `get_mut` methods of the ECC type, but it does so automatically when a read- or write-lock is acquired.

This result shows that our `EccRwLock` type considerably reduces the required code modifications compared to using `RwLock<Ecc<...>>` directly. It still requires a few code adjustments, but mostly mechanical changes that can be easily applied by following the compiler's error messages. In the future, these changes might be even automated using tools such as `cargo fix` [84] or refactoring support of integrated development environments (IDEs).

5.2.2 Robustness

To evaluate the robustness of the `Ecc` wrapper type, we randomly injected bit-flips into the items of the `ContextList` collection while running a test program. We analyzed the progress of the test program both with an original `ContextList` type and a modified `ContextList` type that uses the `EccRwLock` to add an ECC-protection to its items and compared the results.

For analyzing a test run, we monitored the debug output to determine how far the program was able to run before a fatal error occurred. We did multiple test runs with different intervals between bit-flips. To get statistical relevant results, we repeated each test run 100 times and report the median values together with the lower and upper quartiles.

Targeted Fault Injection

The fault injection framework described in Section 5.1.2 allows the injection of bit-flips at arbitrary memory locations. In order to get meaningful results, we injected bit-flips directly into the ECC-protected items of the `ContextList` instead of distributing the bit-flips over the complete memory. This targeted approach has several advantages. First, it eliminates noise from unrelated system crashes, for example due to critical bit-flips in page tables. Second, it considerably increases the probability that a protected item is affected by a random bit-flip. In contrast, flipping a random bit of the complete memory would affect a protected value only with a very low probability.

To be able to perform targeted fault injection, we preallocated a continuous physical memory region as backing storage for the `Context` items of the `ContextList`. We modified the context creation routines to allocate new items from this memory region instead of using heap allocations. This way, our kernel has a statically known physical memory range that stores exactly the ECC-protected `Context` instances that are referenced in the `ContextList`.

By retrieving the physical address and size of this memory region through the debug output of the kernel, we were able to create a fault injection script that injects random bit-flips directly into the protected items of the `ContextList`.

The Test Program

The goal of the test program is to frequently modify the `ContextList` data structure, both by spawning processes and threads and by sending inter process messages through pipes, which are stored inside the `ContextList` too.

A simplified version test program is shown in pseudo-code in Listing 5.4. The program launches 30 child processes that all echo their standard input back as standard output. Then the parent process spawns one thread per child that tests whether the child behaves correctly by sending messages and verifying the responses. Each thread sends 100 message blocks that each contain 10 messages. In total over all threads and processes, this results in 3000 message blocks that are sent and acknowledged. Additionally, each child is stopped with a special *exit* message (not shown in the code except), which results in a total of 3030 message blocks.

While the code except uses only integers as messages, the real program embeds these numbers into short strings to increase the message size. The whole response string is checked for consistency.

Results

Figure 5.1 compares the number of acknowledged message blocks before the test program crashed for different bit-flip intervals with and without ECC protection. The results are shown as a box-and-whisker plot that uses the following conventions: The colored boxes represent the range between the lower and upper quartile, the horizontal line shows the median, and the vertical line (the *whiskers*) show the full range of the data without outliers.

For bit-flip intervals of 100ms and above, almost all runs of the test program finish successfully when the ECC protection of the context list is active. Without it, the lower quartile of acknowledged messages is up to 39% lower. This shows that the `EccRwLock` is able to effectively protect the context list data structure from bit flips.

```
fn main() {
    if is_child() { child() } else { parent() }
}

// echo messages sent on stdin back on stdout
fn child() {
    for message in io::stdin().read() {
        io::stdout().write(message)
    }
}

fn parent() {
    let childs = spawn_childs(); // spaws 30 child processes
    for c in childs {
        thread::spawn(|| test_child(c));
    }
}

fn test_child(c: Child) {
    let next_send = 0; let next_ack = 0;
    for iteration in 0..100 {
        for message_number in 0..10 {
            c.stdin.write(next_send);
            next_send += 1;
        }
        for message_number in 0..10 {
            assert_eq!(c.stdout.read(), next_ack);
            next_ack += 1;
        }
    }
}
```

Listing 5.4: Simplified Test Program in Pseudo-Code

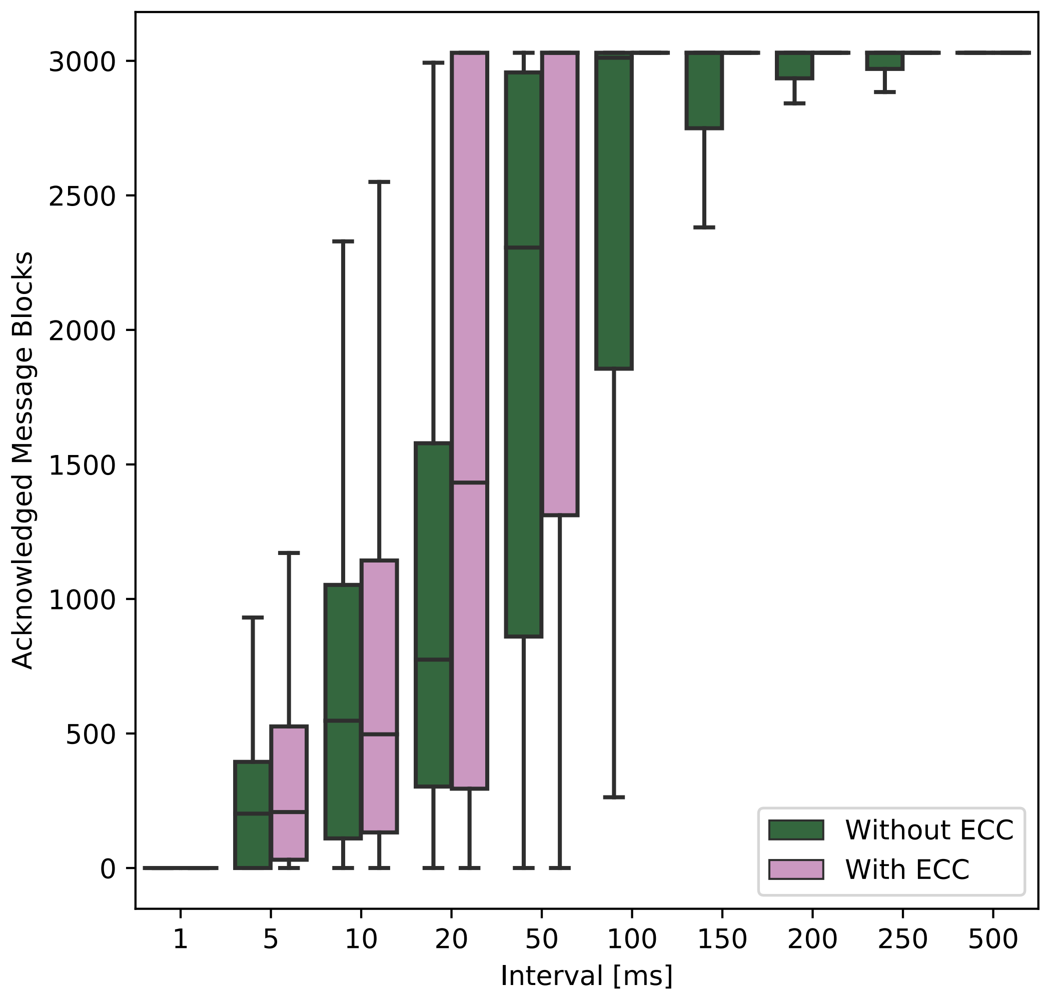

Figure 5.1: Number of Acknowledged Message Blocks Before a Program Crash for Different Bit-Flip Intervals

For an interval of 500ms, almost all runs of the test program finish successfully, even without ECC protection. The reason is that the total number of flipped bits over the program run time is relatively small, so that the probability that a critical bit is flipped is very low.

For an bit-flip interval of 1ms, the program (or the kernel) crashes almost immediately for both enabled and disabled ECC protection. The median and the lower and upper quartiles are 0 for both cases. The reason that the ECC can not effectively protect the context list in this case is probably that the bit-flips occurred after a modified value was written, but before the checksum was recalculated. Also, the used checksum is only able to recover the original value if the copy is still consistent.

The ECC-protection is most effective for bit-flip intervals between 20 and 250ms. It often considerably improves the upper quartile, for example from 1579 to 3030 message blocks for an interval of 20ms. For intervals greater than 50ms, it also improves the lower quartile considerably, for example from 1856 to 3030 message blocks for an interval of 100ms.

Discussion

In summary, the results show that the `EccRwLock` wrapper type considerably improves the resilience against bit-flips for bit-flip intervals of 5ms or larger. For intervals of 100ms or larger, the ECC protections allows the program to finish successfully in almost all cases.

If maximal protection against bit-flips is desired, the `EccRwLock` might not be the best choice as the `RwLock` interface can lead to relatively long periods between writing a value and updating the checksum. The reason is that the checksum is only recalculated after the lock is released again, which can be a relatively long time, for example when a lock is held for the duration of a loop. For improved protection, the explicit interface of the `Ecc` wrapper type can be used to update the checksum immediately after writing a value.

5.2.3 Performance Overhead

To evaluate the performance overhead of our `Ecc` wrapper type, we performed a microbenchmark that measures the execution time of read and write operations on ECC-protected arrays of different sizes. For running the benchmarks, we used the built-in *bench* framework [85] of Rust.

```
#[bench]
fn test_benchmark(b: &mut Bencher) {
    let s = "Hello";
    b.iter(|| {
        let upper = s.to_uppercase();
        test::black_box(upper);
    })
}
```

Listing 5.5: Example Benchmark using the Rust Benchmark Framework

Rust's Benchmark Framework

The *bench* framework supports automated running of microbenchmarks. It repeatedly invokes specified function and measures the average execution time and the variance. The bench framework is shipped with Rust, but still limited to nightly compilers because the exact design is not stabilized yet. The Rust project itself uses the tool for evaluating the performance of the compiler, so even though the tool is not officially considered stable yet, it is well tested in practice.

Listing 5.5 shows an example benchmark function that uses the bench framework. The `#[bench]` attribute marks the function as a benchmark function, so that it can be run by the benchmark runner. All benchmark functions take a `Bencher` instance as argument, which provides an `iter` method that should be used to run the actual benchmark. Everything outside the `iter` method is considered initialization or deinitialization code that should not be measured.

The example function first creates a `"Hello"` string as an initialization routine. Then it calls the `iter` function with a closure that contains the code that should be benchmarked. In this case, a closure that calls the `to_uppercase` method on the string `s` is passed. The bench framework runs the given function multiple times and measures the average time per iteration and the variance. Instead of running the function a fixed number of times, the bench framework dynamically repeats the function passed to `iter` until its runtime converges to a stable median. It also takes care of removing outliers.

The purpose of the `black_box` function is to prevent certain compiler optimizations. For example, without the `black_box` function the compiler could completely remove the `to_uppercase` call because the result is not used. The `black_box` function prevents this by being opaque to the compiler, which means that the compiler has to treat the `upper` variable as used and thus needs to actually execute the `to_uppercase` function.

To execute the benchmark, the built-in `cargo bench` command [86] can be used. This results in the output shown in Listing 5.6. We see that the median runtime of a `to_uppercase` call for `"Hello"` is 109 nanoseconds. The `+/- 1`

```
test_benchmark          ... bench:         109 ns/iter (+/- 1)
```

Listing 5.6: Result of Example Benchmark

```
#[bench]
fn read_array(b: &mut Bencher) {
    let array = black_box([0x2222222u64; ARRAY_SIZE]);
    b.iter(|| {
        for i in 0..ARRAY_SIZE {
            black_box(array[i]);
        }
    })
}
```

Listing 5.7: Implementation of the *Read* Benchmark

means that the variance across the test runs was one nanosecond.

Benchmark Functions

To evaluate the performance of the `Ecc` wrapper type, we implemented 8 benchmark functions. All functions follow the same scheme: They create an array, either with or without ECC protection, and then measure the time for a certain access operation. We implemented the following benchmark functions:

- **Read:** To get a baseline for comparison, this benchmark measures the time needed for reading every element of an unprotected array. The implementation of the function is shown in Listing 5.7. It uses the `black_box` function on the initialized array to prevent constant propagation optimizations. The `black_box` function here behaves like the identity function, i.e. it simply returns the value that was passed as argument. To ensure that no read operations are optimized out, each array element is also passed to the `black_box` function.

- **Write:** This benchmark provides a baseline for write operations. Listing 5.8 shows the implementation of the function. Similar to the read benchmark, it creates an unprotected array wrapped in a `black_box` call. It then overwrites each element with a new value. To prevent optimizations, both the new value and the resulting array are passed to the `black_box` function again.

```
#[bench]
fn write_array(b: &mut Bencher) {
    let mut array = black_box([0x2222222u64; ARRAY_SIZE]);
    b.iter(|| {
        for i in 0..ARRAY_SIZE {
            array[i] = black_box(0x11111111);
        }
        black_box(&array);
    })
}
```

Listing 5.8: Implementation of the *Write* Benchmark

```
#[bench]
fn ecc_array_get_ref(b: &mut Bencher) {
    let array = black_box(Ecc::new([0x2222222u64; ARRAY_SIZE]));
    b.iter(|| {
        black_box(array.get_ref());
    })
}
```

Listing 5.9: Implementation of the *ECC get_ref* Benchmark

- **ECC get_ref:** This benchmark uses an ECC-protected array and measures the time for calling the `get_ref` method of the `Ecc` wrapper type, which verifies the ECC checksum. Listing 5.9 shows the implementation. The `Ecc::new` function is used outside of the `iter` function, so it is not measured.

- **ECC get_mut:** Similar to the *get_ref* benchmark, this benchmark measures the runtime of the `get_mut` function. Apart from calling `get_mut` instead of `get_ref`, the implementation is identical to the `get_ref` benchmark.

- **ECC Read Batched:** This benchmark measures the time to read all elements of an ECC-protected array. Instead of verifying the ECC checksum before reading each element, the checksum is only verified once before the loop. This way, multiple ECC calculations are performed in a single *batch*. Listing 5.10 shows the implementation of the benchmark. The `get_ref` method is called inside the `iter` function, so it is included in the measured time.

- **ECC Read Unbatched:** In contrast to the *batched read* benchmark, this benchmark invokes the `get_ref` method for each array element instead of creating an `array_ref` binding outside of the `for` loop.

```
#[bench]
fn read_ecc_array_batched(b: &mut Bencher) {
    let array = black_box(Ecc::new([0x2222222u64; ARRAY_SIZE]));
    b.iter(|| {
        let array_ref = array.get_ref();
        for i in 0..ARRAY_SIZE {
            black_box(array_ref[i]);
        }
    })
}
```

Listing 5.10: Implementation of the *ECC Read Batched* Benchmark

```
#[bench]
fn write_ecc_array_batched(b: &mut Bencher) {
    let mut array = black_box(Ecc::new(
        [0x2222222u64; ARRAY_SIZE]));
    b.iter(|| {
        let mut array_mut = array.get_mut();
        for i in 0..ARRAY_SIZE {
            array_mut[i] = black_box(0x11111111);
        }
        black_box(&array_mut);
    })
}
```

Listing 5.11: Implementation of the *ECC Write Batched* Benchmark

- **ECC Write Batched:** This function benchmarks the time required for writing to an ECC-protected array through the reference returned from the `get_mut` method. Similar to the *ECC Read Batched* benchmark, this function only invokes `get_mut` once instead of calling it for each array element. The implementation of this function is shown in Listing 5.11.

- **ECC Write Unbatched:** This benchmark is identical to the *ECC Write Batched* benchmark, with the difference that it invokes `get_mut` for every array element, thereby minimizing the time between writing new values and recalculating checksums.

Instead of using a fixed array size, we repeated the benchmarks for different array sizes between 5 and 1000 elements (in steps of 5 elements). The benchmarks were run using QEMU 3.1.0 on an Ubuntu 19.04 system with an Intel Core i5-7300U CPU and 8GB main memory.

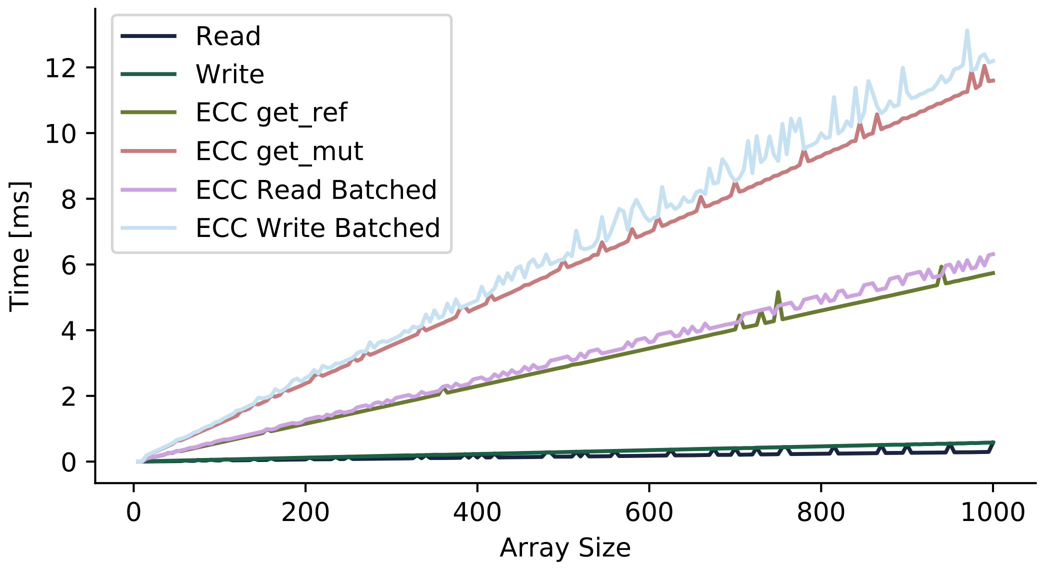

Figure 5.2: Time for accessing all Elements of an Array, with and without ECC Protection

Results

The results of our benchmarks are shown in Figures 5.2 and 5.3. Both graphs plot the array size on the x-axis and the median execution time in milliseconds on the y-axis. The reason that we split the results across two figures is that the *unbatched* ECC tests have a much higher runtime than the other tests.

Figure 5.2 shows the runtime of all tests except the unbatched ECC tests. The results show the following:

- The time for a linear read of the unprotected array grows linearly with the array size, from `0.003ms` for an array size of 10 to `0.294ms` for an array size of 1000.
- Linearly writing the unprotected array takes about twice the time than a linear read. It also grows linearly with the array size, ranging from `0.006ms` for a size of 10 to `0.579ms` for a size of 1000.
- Calling `get_ref` on the array to verify its consistency takes about 19 times as long as a linear read operation (median: `19.02`, lower quartile: `17.57`, upper quartile: `19.32`). Like the linear read, it grows linearly with the array size, which makes sense since it calculates a sum over all array items.
- The `get_mut` method, which additionally recalculates the ECC checksum, takes about twice the time than the `get_ref` method. Compared to an

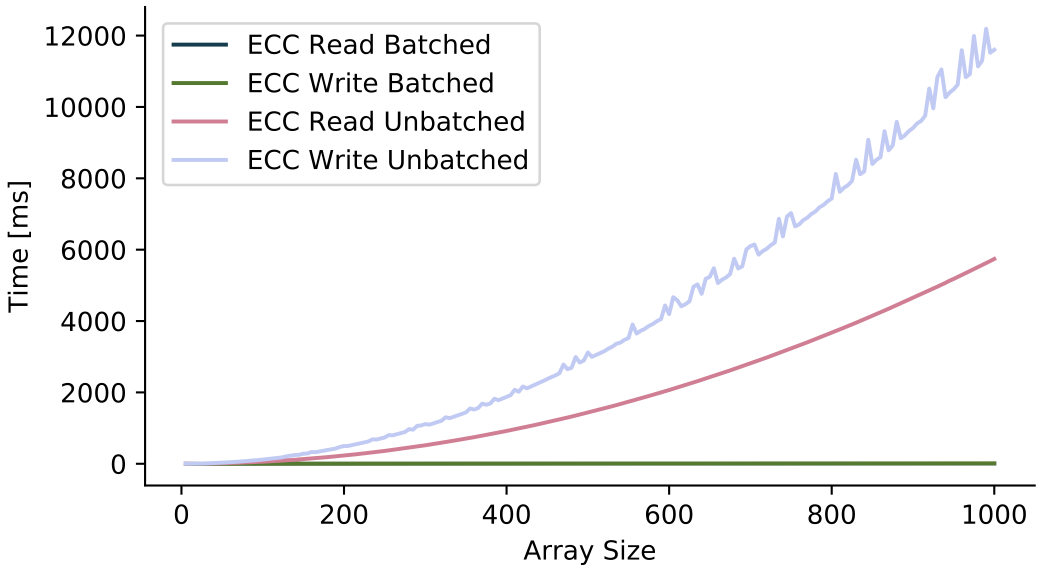

Figure 5.3: Time for Accessing all Elements of an ECC-Protected Array, with and without Batching of ECC calculations

unprotected write operation, it is about 20 times slower (median: `20.12`, lower quartile: `19.99`, upper quartile: `20.46`).

- Batched ECC read operations, which call `get_ref` once and then read all array elements through the returned reference, are only marginally slower than the `get_ref` call alone (factor `1.0578`).
- Similarly, batched ECC write operations are only about `1.0570` times slower than the `get_mut` call alone.

The results indicate that the performance overhead comes mainly from the `get_ref` and `get_mut` methods, which is expected since they are performing the checksum verifications and recalculations. The results from the unbatched ECC tests shown in Figure 5.3 confirm this theory: Calling `get_ref` or `get_mut` on every array element leads to an exponential performance overhead. The reason is that the methods always calculate the checksum on the whole array. Thus, a doubled array size leads to a doubled number of checksum calculations.

The drastically lower runtime of the batched ECC calculation shows a fundamental advantage of our non-transparent ECC design: The programmer is able to manually optimize the performance of ECC calculations through techniques like batching. The test results show that manual batching can lead to drastic performance improvements.

While the general overhead of our `Ecc` wrapper type might seem very high, it is worth noting that the results only show the worst-case overhead for a synthetical microbenchmark. In practical scenarios, the overhead is expected to be much lower. Also, we did not perform any performance profiling or optimization on the `Ecc` type, so we expect that the performance can be further improved, for example through techniques like loop unrolling or utilizing SSE operations (if available).

5.3 Persistent File Descriptors

To evaluate how our framework can be used in a microkernel based operating system both at the userspace and kernel level, we implemented support for persistent file descriptors in Redox OS. In contrast to normal file descriptors, persistent file descriptors are stored in a separate data structure and stay valid across reboots.

5.3.1 Implementation

To implement persistent file descriptors, we modified the kernel to add a new `O_PERSISTENT` flag to the `open` system call. When this flag is given, the kernel does not use the `ContextList` data structure for creating the file descriptor, but instead uses a separate list stored in a `PERSISTENT_FILES` static variable that is annotated with the `#[restorable(restore = true)]` attribute. Note that Redox, while being a microkernel, still stores a sort of file descriptor at kernel level that map a process-specific file descriptor number to a *scheme* (e.g. `filesystem` or `network`) and a scheme-internal file descriptor number.

In the userspace filesystem driver, we did a similar thing: If the `open` operation of the `filesystem` scheme is invoked with the `O_PERSISTENT` flag, the file descriptor is not created in the normal file list, but in a in a `PERSISTENT_FILES` static variable that is annotated with the `#[restorable(restore = true)]` attribute.

5.3.2 Test Program

To evaluate the persistent file descriptor functionality, we created a small test program that uses a persistent static variable to store a file descriptor. A simplified version of program is shown in pseudo-code in Listing 5.12. On the first run, the static variable is `None`, so that a new file descriptor is created through the `open` system call. By passing the `O_PERSISTENT` flag, we tell the kernel and the filesystem driver that the file descriptor should stay valid across reboots.

On subsequent runs, the static variable already contains the previous file descriptor because of the `#[restorable]` attribute. In this case, the file is con-

```
#[restorable(restore = true)]
static FILE_DESCRIPTOR: Mutex<Ecc<Option<RawFd>>> =
    Mutex::new(Ecc::new(None));
const FLAGS = O_CREAT | O_TRUNC | O_WRONLY | O_PERSISTENT;

fn main() {
    println!("test start"); // for debugging
    let mut file_descriptor = FILE_DESCRIPTOR.lock().get_mut();
    let mut file = match file_descriptor {
        Some(fd) => unsafe { File::from_raw_fd(fd) },
        None => {
            let fd = open("file-descriptor-test.txt", FLAGS);
            *file_descriptor = Some(fd);
            unsafe { File::from_raw_fd(fd) }
        }
    };
    file.write_all(random_bytes()).assert("write failed");
    println!("test finished"); // for debugging
    loop {}
}
```

Listing 5.12: Test Program for Persistent File Descriptors in Pseudo-Code

structed directly from the stored file descriptor so that no `open` system call occurs. After constructing the file descriptor, the program verifies that it is valid by writing some random bytes to it. If the write fails, the program panics.

As the last step, the program enters an infinite loop. The reason is that all allocated resources of the program are deallocated when it finishes, including the persistent file descriptor and the NVRAM chunk used as backing storage for the restorable `FILE_DESCRIPTOR` variable. By entering an infinite loop we prevent this and ensure that the file descriptor stays valid across system restarts.

We add the test program to the initialization script of the system so that it is started automatically on each run. As explained in Section 4.2, our implementation currently relies on a deterministic process start order to guarantee that the same slice of NVRAM is allocated to processes across reboots. By starting the test program at the very end of the system's initialization script, we ensure that the `FILE_DESCRIPTOR` variable uses the same chunk of NVRAM as backing storage after each system restart.

5.3.3 Test Process

To evaluate the functionality and robustness of our persistent file descriptor implementation, we ran the test program while injecting system restarts through the QMP protocol. For this, we used the following approach: First, the system is

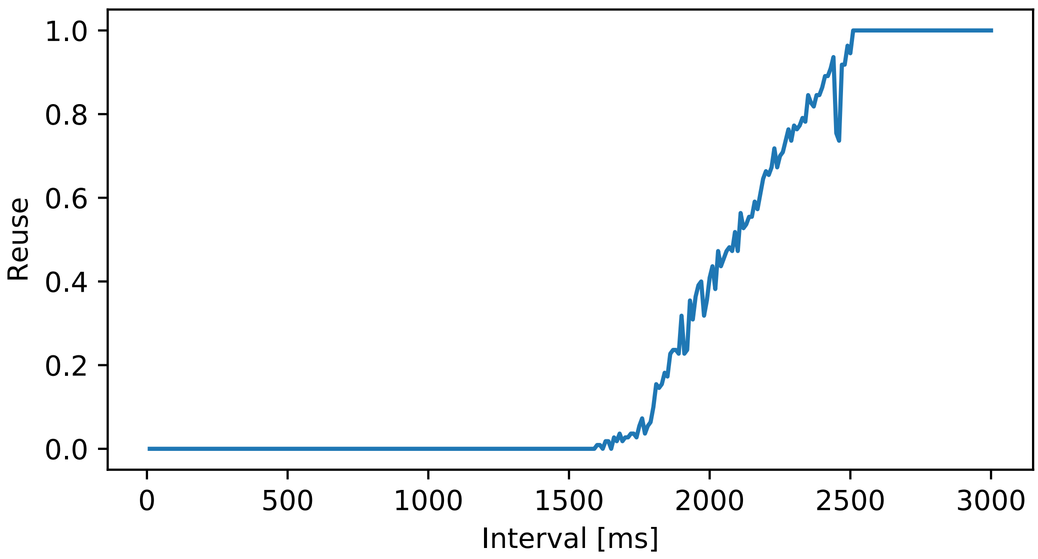

Figure 5.4: Probability That the Test Program Can Reuse the File Descriptors for Different Restart Intervals

started without any previous state. After a specified amount of time, we injected a system restart through QEMU's QMP protocol. Afterwards, we let the system fully boot and record the debug output, which specifies whether a persistent file descriptor from the previous run could be restored. We repeat this process for different restart intervals in steps of 10ms. To get reliable results, we also repeated each run between 10 and 100 times, depending on the variety of the results.

5.3.4 Results

Figure 5.4 shows the probability that the persistent file descriptor can be reused for different reboot intervals. The reason that the results are not fully deterministic is that the QEMU virtual machine is dependent on the host system's scheduling. The results show that the restoration of the file descriptor always succeeds when the time between reboots is greater than about 2.6s. This indicates that the program was always able to run to completion in this time. For shorter reboot intervals, the restoration of the file fails in a greater number of cases because the program was not able to finish the initialization of the restorable `FILE_DESCRIPTOR` variable. For reboot intervals shorter than 1.5s, the program always fails to create the file descriptor before the system restarts.

To evaluate the reliability of persistent file descriptors, we also inspected the debug output for system errors while forcing restarts on the test program. Figure 5.5 shows all errors that we encountered, categorized into different error types.

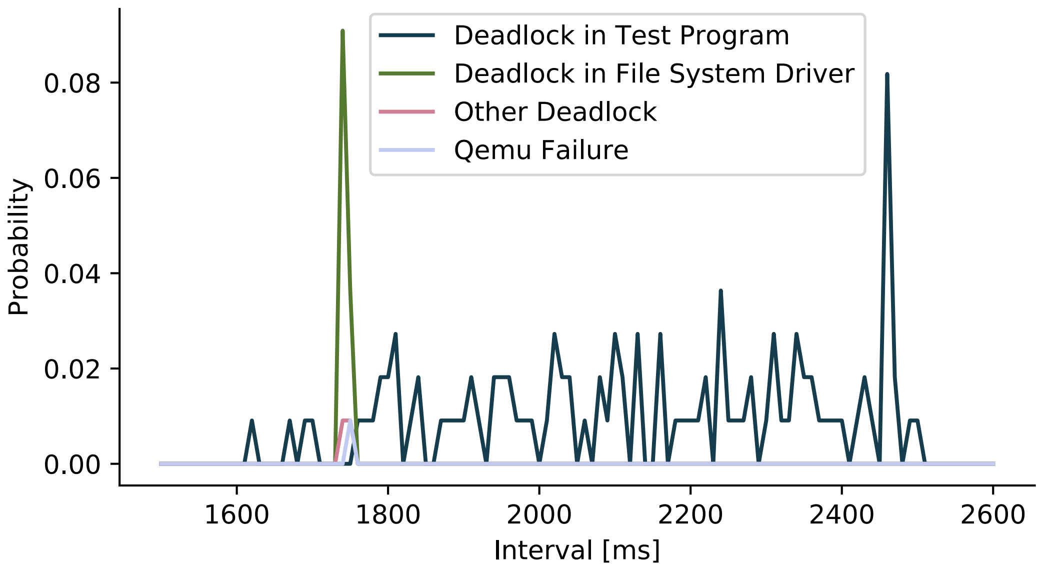

Figure 5.5: Probability of Failures for Different Restart Intervals

We only encountered errors in the "critical" interval range from 1600 ms to about 2550 ms, therefore only this range is shown in the graph. The y-axis specifies the observed probability of the error types over 100 runs.

We observed four different kinds of errors. Most commonly, we saw deadlocks in the test program itself, indicated by the presence of the *test started* message of the test program (see Listing 5.12) in the debug output and no further observable progress of the system. A likely cause for these deadlocks is that the `FILE_DESCRIPTOR` uses a `Mutex`, which remains locked until the `open` system call returns. If the program is interrupted in between, the `Mutex` stays locked so that a deadlock occurs when the program tries to lock it again after the system restart. Future work might be able to resolve this by implementing a way to force-unlock all restorable lock types such as `Mutex` on a system restart.

Apart from deadlocks in the test program, we also observed deadlocks in the file system driver, indicated by debug messages of the driver and no observable system progress afterwards. Interestingly, we only observed file system deadlocks for the restart intervals 1740ms and 1750ms, with probabilities 0.09 and 0.04 respectively. We assume that this is also caused by the acquisition of a restorable lock, probably the lock of the file descriptor collection.

Across all runs, we saw a single deadlock while executing an unrelated part of the system. The deadlock occurred for a restart interval of 1750 ms and caused a system freeze while or after enumerating PCI devices. We do not have a clear explanation for this deadlock. However, we assume that it is related to the kernel routine that initializes a part of the physical address space as NVRAM. This routine currently creates a few restorable static variables for testing purposes, which might cause this deadlock.

Finally, we observed a single test run where QEMU failed to boot after injecting a restart. Only the first line of the initial status message of the BIOS was printed to the debug output, afterwards the execution did not continue. We assume that this might be a bug in QEMU that occurred because the restart command was sent at an unfortunate time (e.g. due to concurrency bugs).

While we experienced some deadlocks, it is worth noting that no other types of system crashes occurred for our test. Notably, we did not observe any kernel panics, system crashes, or other unexpected behavior. Further, no invalid write operations occurred, which indicates that all restored file descriptors were still valid after a system restart. This shows that our prototype implementation is already very robust.

5.3.5 Future Work

While persistent file descriptors are not very useful on their own, they enable other more useful functionality. One example is the use of NVRAM as a file write buffer. The idea is to save all write requests to a persistent file descriptor in a persistent buffer until they are successfully written to disk. If a system restart occurs in between, for example because of a power outage, the requests saved in the buffer can be reapplied after a reboot to prevent data loss.

Chapter 6

Conclusion

The unique combination of persistence, byte-addressability, and high performance makes NVRAM a promising memory technique for a variety of use cases. In this work we explore the use of NVRAM for making microkernel operating systems restartable. For this, we create a framework that allows the safe usage of NVRAM from both userspace applications and operating system kernels.

The framework utilizes the NVRAM as a second, restorable heap area. This way, support for restartability can be gradually added to existing systems by moving selected state from the normal heap to the restorable heap. Apart from explicit allocation functions for persisting heap variables, our framework also provides a `#[restorable]` attribute for persisting static variables with minimal boilerplate.

Since NVRAM is susceptible to bit-flips, we utilize error correcting codes (ECCs) to ensure the bit-wise consistency of values that live on the restorable heap. Instead of adding ECCs implicitly to values, we give the programmer full control over the use of ECCs through explicit wrapper types. To minimize the required code modifications when introducing an explicit ECC-protection to existing types, we provide ECC-augmented versions of common wrapper types such as `RwLock`. We also provide support for nested consistency checks that recursively verify the checksums of ECC-protected values behind pointers.

By utilizing the type system of the Rust programming language, our framework is able to provide extensive compile-time guarantees. Apart from preventing memory safety errors, which is already ensured by Rust's ownership system, our framework also enforces an ECC-protection for all values stored on the restorable heap. Further, our framework limits the types allowed in NVRAM to guarantee that no dangling pointers can occur due to partially restored state after a system restart.

To evaluate the effectiveness of the ECC-wrapper types provided by our framework, we protected parts of the kernel state of *Redox OS*, which is an experimental microkernel operating system written in Rust. By selectively injecting bit-flips into the protected values through the QMP protocol of the emulator QEMU, we verify that the ECC-protection considerably improves the resilience against bit-flips. Through microbenchmarks, we show that the ECC wrapper types can potentially lead to a high performance overhead. However, it is possible to drastically decrease this overhead by strategically batching checksum verifications and recalculations.

In the second part of our evaluation, we used the NVRAM framework to implement support for persistent file descriptors in Redox. By injecting system restarts through the QMP interface, we show that a test program is able to continue using an opened file across system restarts. By analyzing the errors that occurred over all runs of the test program, we demonstrate the robustness of our implementation.

In summary, we are able to show that our framework enables the safe usage of NVRAM. Through custom attributes and wrapper types, it provides high usability with minimal boilerplate. By persisting only a selected subset of system state, the gradual transformation of existing systems is possible. The prototype implementation of persistent file descriptors, which only provide limited utility on their own, shows that our framework makes it possible to persistent complex system state that involves multiple operating system layers.

Bibliography

[1] Intel. Intel® Optane™ Technology, Aug 2019. [Online; accessed 5. Aug. 2019].

[2] Adrian Proctor. NV-DIMM: Fastest tier in your storage strategy. *Viking Technology whitepaper*, pages 1–7, 2012.

[3] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, Dejan Milojicic, and Vanish Talwar. Using active NVRAM for I/O staging. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, pages 15–22. ACM, 2011.

[4] Tim Allen. Optimizing in-memory databases for advanced analytics. *Intel IT Peer Network*, Aug 2018. [Online; accessed 22. Aug. 2019].

[5] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. Persistent buffer management with optimistic consistency. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, pages 14:1–14:3, New York, NY, USA, 2019. ACM.

[6] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters*, 9(1):526, 2014.

[7] Manzur Gill, Tyler Lowrey, and John Park. Ovonic unified memory-a high-performance nonvolatile memory technology for stand-alone memory and embedded applications. In *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No. 02CH37315)*, volume 1, pages 202–459. IEEE, 2002.

[8] Shyh-Shyuan Sheu, Kuo-Hsing Cheng, Meng-Fan Chang, Pei-Chia Chiang, Wen-Pin Lin, Heng-Yuan Lee, Pang-Shiu Chen, Yu-Sheng Chen, Tai-Yuan Wu, Frederick T Chen, et al. Fast-write resistive RAM (RRAM) for embedded applications. *IEEE Design & Test of Computers*, 28(1):64–71, 2010.

[9] Katherine Bourzac. Has Intel created a universal memory technology? [news]. *IEEE Spectrum*, 54(5):9–10, 2017.

[10] Jason Mick. If Intel and Micron's "Xpoint" is 3D Phase Change Memory, boy did they patent it, Jul 2015. [Online; accessed 7. Aug. 2019].

[11] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004.*, pages 587–590. IEEE, 2004.

[12] Sanmina Corporation. Viking Technology collaborates with Sony Corporation to bring ReRAM storage class memory to NVDIMM markets, Aug 2015. [Online; accessed 23. Aug. 2019].

[13] Zhiqiang Wei, Y Kanzawa, K Arita, Y Katoh, K Kawai, S Muraoka, S Mitani, S Fujii, K Katayama, M Iijima, et al. Highly reliable TaOx ReRAM and direct evidence of redox reaction mechanism. In *2008 IEEE International Electron Devices Meeting*, pages 1–4. IEEE, 2008.

[14] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191):80, 2008.

[15] John Paul Strachan, Antonio C Torrezan, Feng Miao, Matthew D Pickett, J Joshua Yang, Wei Yi, Gilberto Medeiros-Ribeiro, and R Stanley Williams. State dynamics and modeling of tantalum oxide memristors. *IEEE Transactions on Electron Devices*, 60(7):2194–2202, 2013.

[16] Ramtron International Corporation. F-RAM technology brief, 2007.

[17] Takayuki Kawahara, Riichiro Takemura, Katsuya Miura, Jun Hayakawa, Shoji Ikeda, Young Min Lee, Ryutaro Sasaki, Yasushi Goto, Kenchi Ito, Toshiyasu Meguro, et al. 2 Mb SPRAM (SPin-transfer torque RAM) with bit-by-bit bi-directional current write and parallelizing-direction current read. *IEEE Journal of Solid-State Circuits*, 43(1):109–120, 2008.

[18] Yiming Huai et al. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin*, 18(6):33–40, 2008.

[19] David W Abraham and Philip L Trouilloud. Thermally-assisted magnetic random access memory (MRAM), May 7 2002. US Patent 6,385,082.

[20] Peter Vettiger, G Cross, M Despont, U Drechsler, U Durig, B Gotsmann, W Haberle, MA Lantz, HE Rothuizen, R Stutz, et al. The" millipede"-nanotechnology entering data storage. *IEEE Transactions on nanotechnology*, 1(1):39–55, 2002.

[21] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 74–83, New York, NY, USA, 1996. Association for Computing Machinery.

[22] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. Membrane: Operating system support for restartable file systems. *ACM Transactions on Storage (TOS)*, 6(3):11, 2010.

[23] Alex Depoutovitch and Michael Stumm. Otherworld: giving applications a chance to survive os kernel crashes. In *Proceedings of the 5th European conference on Computer systems*, pages 181–194. ACM, 2010.

[24] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 125–130. IEEE, 2001.

[25] George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, and Rakesh Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings international conference on dependable systems and networks*, pages 605–614. IEEE, 2002.

[26] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. *ACM SIGARCH Computer Architecture News*, 40(1):401–410, 2012.

[27] Microsoft Security Response Center. A proactive approach to more secure code, July 2019. [Online; accessed 20. Aug. 2019].

[28] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *ACM SIGPLAN Notices*, 50(4):297–310, 2015.

[29] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. *Commun. ACM*, 54(2):100–107, February 2011.

[30] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P Jouppi, and Mattan Erez. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 466–477. IEEE, 2011.

[31] Jing Li, Binquan Luan, and Chung Lam. Resistance drift in phase change memory. In *2012 IEEE International Reliability Physics Symposium (IRPS)*, pages 6C–1. IEEE, 2012.

[32] Arthur Martens, Rouven Scholz, Phil Lindow, Niklas Lehnfeld, Marc A. Kastner, and Rüdiger Kapitza. Dependable non-volatile memory. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 1–12, New York, NY, USA, 2018. ACM Press.

[33] Bin Gao, Haowei Zhang, Bing Chen, Lifeng Liu, Xiaoyan Liu, Ruqi Han, Jinfeng Kang, Zheng Fang, Hongyu Yu, Bin Yu, et al. Modeling of retention failure behavior in bipolar oxide-based resistive switching memory. *IEEE Electron Device Letters*, 32(3):276–278, 2011.

[34] Jubong Park, Minseok Jo, El Mostafa Bourim, Jaesik Yoon, Dong-Jun Seong, Joonmyoung Lee, Wootae Lee, and Hyunsang Hwang. Investigation of state stability of low-resistance state in resistive memory. *IEEE Electron Device Letters*, 31(5):485–487, 2010.

[35] Shimeng Yu, Yang Yin Chen, Ximeng Guan, H-S Philip Wong, and Jorge A Kittl. A Monte Carlo study of the low resistance state retention of HfOx based resistive switching memory. *Applied Physics Letters*, 100(4):043507, 2012.

[36] CVE-2019-8912. Available from MITRE, 2019. [Online; accessed 20. Aug. 2019].

[37] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. ACM, 2012.

[38] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[39] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[40] Dmitry Vyukov. Syzkaller, 2015. Available at `https://github.com/google/syzkaller`.

[41] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the servo web browser engine using rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 81–89, New York, NY, USA, 2016. ACM.

[42] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. Practical safe linux kernel extensibility. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 170–176. ACM, 2019.

[43] LWN.net. Discussion: Rust is the future of systems programming, C is the new Assembly (Packt), Aug 2019. [`https://lwn.net/Articles/797828` ; accessed 30. Aug. 2019].

[44] Hans-J Boehm. Position paper: nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 9–14. ACM, 2012.

[45] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[46] ISO. ISO/IEC 14882:2017: Programming languages - - C++, 2017.

[47] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[48] Valgrind Developers. Helgrind: a thread error detector, Aug 2019. [Online; accessed 12. Aug. 2019].

[49] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71. ACM, 2009.

[50] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[51] Bill Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.

[52] Google. The Go programming language, Aug 2019. [Online; accessed 12. Aug. 2019].

[53] Google. Frequently asked questions (FAQ) - the Go programming language, Aug 2019. [Online; accessed 12. Aug. 2019].

[54] Manas Technology Solutions. Crystal programming language, Aug 2019. [Online; accessed 12. Aug. 2019].

[55] Manas Technology Solutions. Crystal: Concurrency, Aug 2019. [Online; accessed 12. Aug. 2019].

[56] Luis M Carril and Walter F Tichy. Predicting and witnessing data races using CSP. In *NASA Formal Methods Symposium*, pages 400–407. Springer, 2015.

[57] Google. Data race detector - the Go programming language, Aug 2019. [Online; accessed 12. Aug. 2019].

[58] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. Association for Computing Machinery.

[59] Nicholas D Matsakis and Thomas R Gross. A time-aware type system for data-race protection and guaranteed initialization. In *OOPSLA*, volume 10, pages 634–651, 2010.

[60] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. *ACM SIGPLAN Notices*, 44(1):379–391, 2009.

[61] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[62] James S Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical report, Technical Report UTCS-97-372, 1997.

[63] Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 83–92. ACM, 2012.

[64] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, Piscataway, NJ, USA, June 2013. IEEE Press.

[65] Christoph Borchert and Olaf Spinczyk. Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*, pages 1–7, New York, NY, USA, October 2015. ACM Press.

[66] Tobias Stumpf. How to protect the protector. In *Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015) - Student Forum*, Rio de Janeiro, June 2015.

[67] Muhammad Shafique, Philip Axer, Christoph Borchert, Jian-Jia Chen, Kuan-Hsun Chen, Björn Döbel, Rolf Ernst, Hermann Härtig, Andreas Heinig, Rüdiger Kapitza, et al. Multi-layer software reliability for unreliable hardware. *it-Information Technology*, 57(3):170–180, 2015.

[68] Philippe Forin. Vital coded microprocessor principles and application for various transit systems. *IFAC Proceedings Volumes*, 23(2):79–84, 1990.

[69] Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. Effectiveness of software-based hardening for radiation-induced soft errors in real-time operating systems. In *International Conference on Architecture of Computing Systems*, pages 3–15. Springer, 2017.

[70] Alejandro David Velasco, Bartolomeo Montrucchio, and Maurizio Rebaudengo. TMR technique for mutex kernel data structures. In *2017 18th IEEE Latin American Test Symposium (LATS)*, pages 1–6. IEEE, 2017.

[71] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[72] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[73] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, San Francisco, 2018. Available at `https://doc.rust-lang.org/book/`.

[74] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[75] Felix S Klock II. RFC: Kinds of allocators, Apr 6. [Available at `https://github.com/rust-lang/rfcs/blob/master/text/1398-kinds-of-allocators.md` ; accessed 29. Aug. 2019].

[76] John Ericson. Pull request: Allocator- and fallibility-polymorphic collections, May 2019. [Available at `https://github.com/rust-lang/rust/pull/60703` ; accessed 29. Aug. 2019].

[77] Jorge Aparicio. xargo: The sysroot manager that lets you build and customize std, Aug 2019. [Online; accessed 29. Aug. 2019].

[78] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[79] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.

[80] QEMU Wiki, Feb 2019. [Available at `https://wiki.qemu.org/Main_Page` ; accessed 1. Sep. 2019].

[81] QEMU version 4.1.0 User Documentation, Aug 2019. [Online; accessed 1. Sep. 2019].

[82] QEMU Wiki: Documentation/QMP, Aug 2019. [Online; accessed 1. Sep. 2019].

[83] Xilinx Wiki: QEMU, Sep 2019. [Available at `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842060/QEMU` ; accessed 1. Sep. 2019].

[84] The Cargo Book: fix, Aug 2019. [Available at `https://doc.rust-lang.org/cargo/commands/cargo-fix.html` ; accessed 2. Sep. 2019].

[85] The Rust Book: Benchmark tests, Mar 2017. [Available at `https://doc.rust-lang.org/1.16.0/book/benchmark-tests.html` ; accessed 3. Sep. 2019].

[86] The Cargo Book: bench, Aug 2019. [Available at `https://doc.rust-lang.org/cargo/commands/cargo-bench.html` ; accessed 3. Sep. 2019].